TARTU ÜLIKOOL
MATEMAATIKATEADUSKOND
ARVUTITEADUSE INSTITUUT
TARKVARASÜSTEEMIDE ÕPPETOOL
INFORMAATIKA ERIALA

RISTO VAARANDI

HAJUSAD FAILISÜSTEEMID

MAGISTRITÖÖ

JUHENDAJA: MAG. VILJO SOO

Autor	“.....” ....
Juhendaja	“.....” ....
Õppetooli juhataja	“.....” ....

TARTU
1996

There are always more experiments to be done, more cases to check, more references to search, but eventually, one has to write something down and live with it [Blaze93].

Sissejuhatus

Käesolev töö on pühendatud hajusatele failisüsteemidele. See hajussüsteemide teooria uurimisvaldkond on sedavõrd lai, et magistritöö raames on võimatu täielikkusele pretendeerivat hajusate failisüsteemide käsitlust esitada. Seetõttu tuli teha valik, millist probleemideringi lähemalt vaadelda. Käesolev töö annab üksikasjalikuma ülevaate sellest, kuidas kasutatakse hajusas failisüsteemis vahemälusid. Teisi valdkondi (replikatsioon, süsteemi julgeolek jm.) vaadeldakse põgusamalt. Kuigi see töö on oma iseloomult referatiivne, käsitletakse siin koos teemasid, mida varem pole ühiselt vaadeldud. Mulle teadaolevatel andmetel pole siiani ilmunud ühtegi hajusate failisüsteemide vahemäludele pühendatud monograafiat.

Töö koosneb kolmest peatükist. Esimeses peatükis defineeritakse edaspidises vajalikud mõisted ning esitatakse failiteenuse mudel. Töö teises peatükis tutvustatakse kahte populaarseimat hajusat failisüsteemi. NFS on kõige enam levinud hajus failisüsteem maailmas - peaaegu kõik viimase kümne aasta jooksul erinevate tarkvaratootjate poolt loodud operatsioonisüsteemid sisaldavad NFSi kasutamise tuge. AFSi peetakse üheks kõige skaleeritavamaks (laienemisvõimelisemaks) hajusaks failisüsteemiks, mis on praegu laiemalt kasutusel. Kolmandas peatükis, mis moodustab peaaegu kaks kolmandikku käesolevast tööst, vaadeldakse lähemalt hajusa failisüsteemi vahemäludega seotud probleeme. Otsustasin keskenduda sellele valdkonnale, kuna vahemälude kasutamist peetakse hajusate failisüsteemide puhul kohustuslikuks. Tuntuim hajusate failisüsteemide uurija M. Satyanarayanan on öelnud: *Pole kahtlust, et vahemälud mängivad hajusa failisüsteemi jõudluse tõstmisel kõige suuremat rolli. Tänapäeval ei leidu ühtegi tõsiseltvõetavat hajusat failisüsteemi, kus vahemälusid ei kasutataks* [Sat93].

Selle töö kirjutamisel aitasid mind otseselt rohkem kui 30 allikat - 6 doktoriväitekirja, 2 monograafiat ning suurem hulk artikleid ja tehnilisi raporteid, mis pärinevad aastatest 1985-1996 (vt. kasutatud kirjanduse loetelu). Olgu siinkohal öeldud, et peaaegu kõik tehnilised raportid on avaldatud ka artiklitena. Ühest küljest võimaldas allikate rohkus tööd sisukamaks muuta, teiselt poolt muutis see töö kirjutamise raskemaks. On üldiselt teada, et hajusate failisüsteemide teoorias on paljudel terminitel mitu tähendust, milles ma ka ise kasutatud kirjandust lugedes

veendusin. Tegin oma parima, et erinevate autorite ideid ja mõtteid ühtseks käsitluseks vormida. Loodan, et see töö mõjub terviklikult ning laiendab lugeja silmaringi.

Sisukord

1. Hajussüsteem ja hajus failisüsteem

1.1 Mõisted

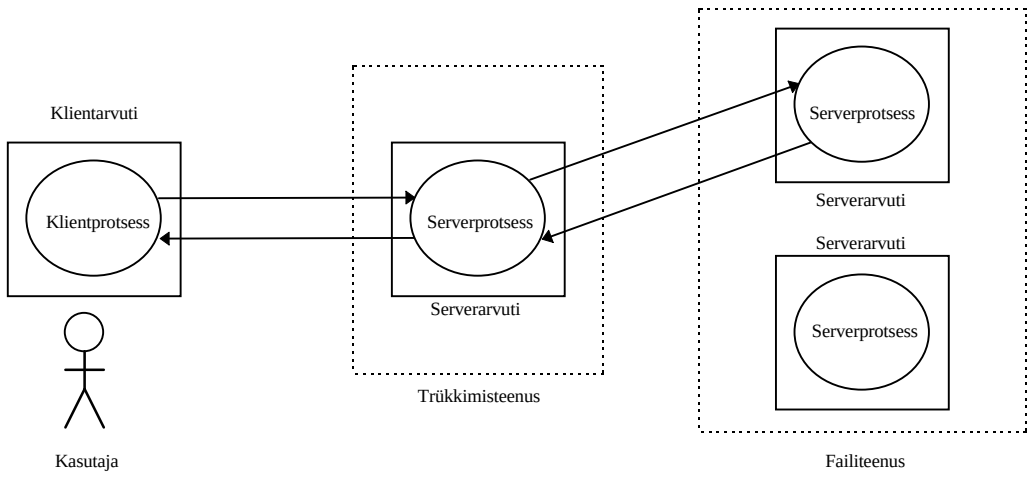
Hajussüsteemiks (*distributed system*) nimetatakse hajussüsteemi tarkvaraga varustatud sõltumatute arvutite ehk *sõlmede* (*nodes*) hulka ning neid sõlmi ühendavat arvutivõrku. Hajussüsteemi tarkvara võimaldab sõlmedes töötavatel protsessidel koordineerida üksteise tegevust ja jagada hajussüsteemi ressursse - andmeid ning riist- ja tarkvara. Näiteks kohtvõrk ning selle abil ühendatud arvutid, millest ühe juurde paigutatud laserprinterit kasutavad teistes arvutites töötavad protsessid, moodustavad hajussüsteemi.

Ressursiks (*resource*) nimetatakse riistvara, tarkvara või andmeid (näiteks printerit, programmi originaali või andmebaasi kirjet), mis on paigutatud ühe hajussüsteemi arvuti juurde. Mingi arvuti juures paikneva ressurssi jagamiseks peab selles arvutis töötama ressurssi haldav protsess ehk *ressursihaldur* (*resource manager*), mis võimaldab teistes sõlmedes töötavatel protsessidel ressurssi üle arvutivõrgu kasutada. Ressursile on võimalik ligi pääseda ainult ressursihalduri vahendusel (ka siis, kui ressurssi kasutab selle haldajaga samas sõlmes töötav protsess). Tihti haldab protsess mitut ressurssi (näiteks jagatava andmebaasi kõiki kirjeid).

Iga hajussüsteem on üles ehitatud teatud mudeli kohaselt. Populaarseima *klient-server mudeli* (*client-server model*) korral koondatakse jagatavad ressursid üksikute sõlmede juurde ning iga sellist ressurssi haldab *serverprotsess*. Kui mingi protsess, mida nimetatakse *klientprotsessiks*, vajab teatud ressurssi, pöördub ta sellega opereerimiseks vastava serverprotsessi poole. Pärast nõutud operatsiooni ressursile rakendamist tagastab serverprotsess klientprotsessile vastuse, kui selline on operatsiooni iseloom või kui klientprotsess nõuab kinnitust operatsiooni täitmise kohta. Edaspidi nimetatakse klient- ja serverprotsessi mõnikord lihtsalt *kliendiks* ja *serveriks*. Nende terminite tähendus sõltub tegelikult kontekstist, sest erialases kirjanduses mõistetakse nende all tihti ka sõlmi, kus vastavad protsessid töötavad. Näiteks failiserver võib tähendada nii serverprotsessi, mis haldab jagatavaid faile, kui ka arvutit, kus töötab serverprotsess ja mille kettal need failid asuvad.

Teenuseks (*service*) nimetatakse ressursihalduri poolt teistele protsessidele pakutavaid võimalusi mingi ressursi (mingite ressursside) kasutamiseks. Klient-server mudeli korral nimetatakse teenust kasutavat klientprotsessi tihti antud *teenuse kliendiks* ning teenust pakkuvat serverprotsessi antud *teenuse serveriks*. Uue teenuse loomisel hajussüsteemis defineeritakse selle *liides*, kus kirjeldatakse kõik lubatud operatsioonid teenuse kasutamiseks - operatsioonid, mida on võimalik ressursihalduri vahendusel ressursile rakendada. Võib juhtuda, et mitu erinevates sõlmedes töötavat ressursihaldurit haldavad üht tüüpi ressursse ning pakuvad ülejäänud protsessidele ühtset teenust. Näiteks failiteenuse korral võivad seda teenust pakkuda kaks kataloogipuu erinevaid harusid haldavat protsessi.

Enamasti on hajussüsteemis ressursse tarbivateks protsessideks kasutajate poolt käivitatud programmid, mille vahendusel saavad kasutajad ise hajussüsteemi ressursse tarbida. Samas võib mingit teenust pakkuv ressursihaldur omakorda olla mingi teise teenuse kasutaja. Näiteks klient-server mudeli korral võib trükkimisteenuse server oma töö käigus tekkivate ajutiste failide paigutamisel kasutada failiteenust (vt. Joonis 1.1).



Joonis 1.1 Trükkimisteenuse server on failiteenuse klient. Klientide ja serverite vaheline kommunikatsioon on tähistatud nooltega.

1.2 Kaugprotseduurid

Hajussüsteemi teenuseid on mugav kasutada *kaugprotseduuride väljakutsete* (*remote procedure calls*) abil. *Kaugprotseduuriks* nimetatakse protseduuri, mille keha ei asu mitte protseduuri väljakutsuvas protsessis, vaid mõnes muus, üldjuhul teises sõlmes töötavas protsessis. Klient-server mudeli korral on loomulik teenuse liideses kirjeldatud operatsioonid realiseerida kaugprotseduuridena, mida kutsutakse välja klientprotsessidest ning mille kehad asuvad serverprotsessis.

Kaugprotseduuride realiseerimist võimaldavate vahendite väljatöötamisel eeldatakse harilikult, et on juba olemas *teateedastusprimitiivid* (*message passing primitives*) suvalisi andmeid sisaldava *teate* (*message*) saatmiseks üle võrgu teisele protsessile ning sellise teate vastuvõtmiseks. Teateedastusprimitiivid on enamasti realiseeritud operatsioonisüsteemi tasemel (näiteks BSD UNIXis süsteemifunktsioonidena).

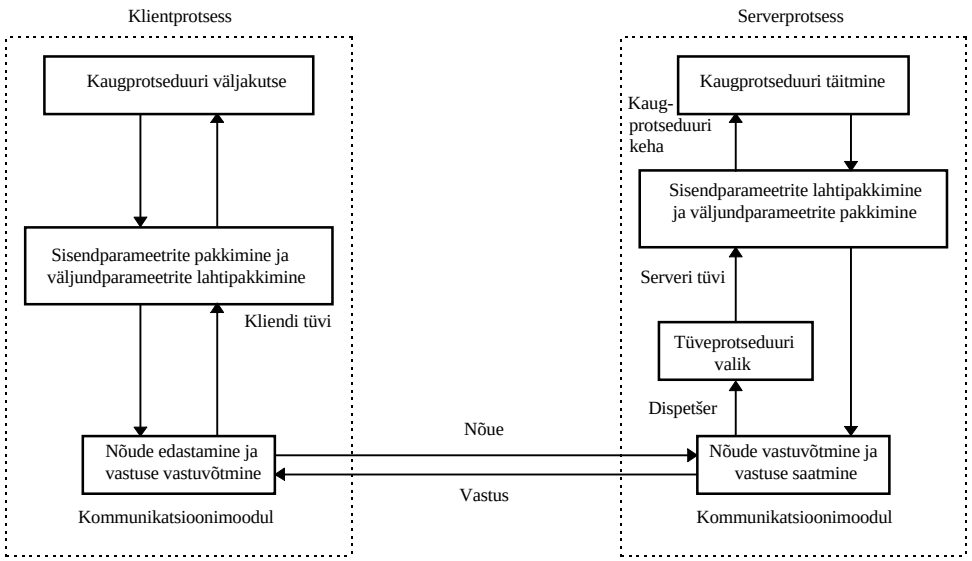
Kaugprotseduuri väljakutsumisel pakitakse sisendparameetrid nõudeteatesse ehk *nõudesse* (*request*) ja saadetakse serverprotsessile, kusjuures serverprotsess tagastab väljundparameetrid vastusteatega ehk *vastusega* (*reply*). Vastuse saatmine on otstarbekas ka juhul, kui väljundparameetrid tegelikult puuduvad, sest vastuse saamisest saab kaugprotseduuri väljakutsuja järeldada, et nõue jõudis kohale ja protseduur täideti.

Vahendid kaugprotseduuride realiseerimiseks võivad sisalduda programmeerimiskeeles (Cedar, Argus jt.), nende puudumisel tuleb kasutada erilist *liidese defineerimise keelt* (*interface definition language*) koos vastava kompilaatoriga. Liidese definitsiooniga antakse serverprotsessis sisalduvate kaugprotseduuride *signatuurid* (*signatures*), millest igaüks koosneb protseduuri nimest ning sisend- ja väljundparameetrite tüüpidest.

Liidese definitsiooni kompileerimisel saadakse kaks tüve - klient- ja serverprogrammi koosseisu kuuluvad erilised tarkvaramoodulid. *Tüvi* (*stub*) koosneb põhiliselt *tüveprotseduuridest* (*stub procedures*), kus igale kaugprotseduurile vastab üks tüveprotseduur. Klient- või serverprogrammi saamiseks lingitakse programmi traditsioonilises keeles kirjutatud osad vastavalt *kliendi* või *serveri tüvega* ning teegina etteantud protsessidevahelist suhtlemist võimaldava *kommunikatsioonimooduliga*.

Kliendi tüvi sisaldab ainult tüveprotseduure, mille ülesandeks on kaugprotseduuride parameetrite pakkimine nõudesse, nõude üleandmine kommunikatsioonimoodulile, mis selle serverprotsessile edastab, ja kommunikatsioonimoodulilt serverprotsessi vastuse ootamine ning lõpuks vastuses sisalduvate väljundparameetrite lahtipakkimine ja tagastamine.

Serveri tüvi koosneb tüveprotseduuridest ja *dispetšerist* (*despatcher*), mille poole pöördutakse vahetult kommunikatsioonimoodulist pärast nõude vastuvõtmist. Dispetšer edastab saadud nõude õigele tüveprotseduurile. Tüveprotseduuride ülesanne on nõudes sisalduvate parameetrite lahtipakkimine, kaugprotseduuride tegelik väljakutsumine, seejärel väljundparameetrite vastuvõtmine, nende pakkimine vastusesse ning vastuse üleandmine kommunikatsioonimoodulile.



Joonis 1.2 Kaugprotseduuri väljakutse.

Kommunikatsioonimooduli ülesandeks on protsessidevahelise suhtlemise korraldamine vastavalt kaugprotseduuri *väljakutsesemantikale*. Näiteks *vähemalt-üks-kord-semantika* (*at-least-once-semantics*) tagamiseks võib klientprotsessipoolne kommunikatsioonimoodul nõude uuesti edastada, kui teatud aja möödumisel pole vastust saabunud. Rangema *täpselt-üks-kord-semantika* (*exactly-once-semantics*) saavutamiseks peab serverprotsessipoolne kommunikatsioonimoodul kasutama *ajaloomehhanismi* - jätma meelde kord juba tagastatud vastused ning nõude korduval

saamisel tagastama eelnevalt salvestatud vastuse, ilma et kaugprotseduuri uuesti täidetaks.

Siin kirjeldatud skeem klient- ja serverprotsessi vaheliseks kommunikatsiooniks on küllalt levinud spetsiifilise iseloomuga teenuste korral. Paraku ei sobi see üldise iseloomuga laiemaks kasutamiseks mõeldud teenuse jaoks, sest tihti on seda teenust vaja kasutada ka programmidel, mis on kirjutatud enne selle teenuse loomist. Nimelt tuleb sellisel juhul neid programme modifitseerida, et nende tekstides pöördutaks teenuse liideses kirjeldatud operatsioone realiseerivate kaugprotseduuride poole. Seejärel tuleb muudetud programmid uuesti kompileerida ning kliendi tüve ja kommunikatsioonimooduliga linkida. Selline lahendus on ebaotstarbekas, kui muutmist vajavaid programme on palju. Näiteks failiteenuse korral pole see mõeldav, sest peaaegu iga programm tegeleb mingil määral failitöötlusega.

Seetõttu võetakse tihti kasutusele klientarvuti protsessina või klientarvuti operatsioonisüsteemi osana realiseeritud *kliendimoodul* (*client module*), kus toimuvad vahetud pöördumised kaugprotseduuride poole. Peale selle modifitseeritakse klientarvuti operatsioonisüsteemi, et teatud süsteemifunktsioonide poole pöördumisel pöördutaks omakorda kliendimooduli poole. Näiteks failiteenuse korral modifitseeritakse failitöötlusega seotud süsteemifunktsioone.

1.3 Hajusa failisüsteemi mõiste ja sellele esitatavad nõuded

Esimeste hajussüsteemide loomine sai põhimõtteliselt võimalikuks kiirete kohtvõrgutehnoloogiate ja esimeste tööjaamade loomisega 1970-ndatel aastatel. Tolle aja olulisimad uuringud toimusid Xeroxi Palo Alto uurimiskeskuses (Xerox PARC), mis rajasid aluse hajussüsteemide edasisele arengule. Ajavahemikus 1971-1980 loodi seal esimene kiire kohtvõrgutehnoloogia (Ethernet), konstrueeriti esimesed tänapäeva tööjaamade eelkäijad (Alto) ning loodi hulk eksperimentaalseid hajussüsteeme. Esimeseks selliseks süsteemiks oli hajus failisüsteem XDFS [Coul94]. Ka tänapäeval, mil hajussüsteeme kasutatakse ka väljaspool laboratooriume, on suur osa hajussüsteemidest kas puhtakujulised hajusad failisüsteemid (näiteks NFS) või sisaldavad endas failiteenust (näiteks Sprite).

Hajus failisüsteem (*distributed file system*) on hajussüsteem, kus leidub failiteenus. Kuna sageli on hajus failisüsteem ehitatud üles klient-server mudelile, koosneb ta tüüpiliselt paljudest klientarvutitest, ühest või mitmest serverarvutist, mille kettad (või muud liiki püsimälu) on piisavalt suure mahtuvusega, ning arvutivõrgust, mille kaudu klientarvutites töötavad protsessid kasutavad serverarvutite ketastel asuvaid faile. Terminit *hajus failisüsteem* kasutatakse edaspidi peale konkreetse installatsiooni tähistamise ka hajussüsteemi *liigi* tähenduses. Hajusa failisüsteemi nimi (NFS, AFS, Sprite vms.) võib tähistada nii hajusa failisüsteemi liiki kui antud liiki hajusa failisüsteemi tarkvara.

Hajusate failisüsteemide populaarsusel on palju põhjusi. Nad pakuvad andmete jagamise võimalust - üks kasutaja pääseb vajadusel ligi teise andmetele. Samuti võib suvaline kasutaja töötada oma andmetega süsteemi erinevate sõlmede juures. Kasutajatel on võimalik normaalseks tööks kasutada ka väikese kettamäluga sõlmi, sest kõiki tööks vajalikke faile ei pea enam lokaalsel kettal hoidma. Klient-server mudeli korral paikneb enamus faile süsteemi üksikute sõlmede (serverarvutite) juures ning administreerimisega (varukoopiate tegemine, uue tarkvara installeerimine jms.) tegelevad süsteemiadministraatorid, vabastades lihtkasutajad sellest ülesandest.

Selleks, et hajus failisüsteem oleks tõeliselt efektiivne, peab ta olema võimalikult *läbipaistev* (*transparent*) [Coul94]. Tähtsaim läbipaistvuse liik on süsteemi ressurssidele *ligipääsu läbipaistvus* (*access transparency*) - protsessid saavad nii lokaalseid faile kui mittelokaalseid ehk *kaugeid faile* (*remote files*) kasutada ühesuguste operatsioonide abil, kõik algselt lokaalse failisüsteemi jaoks kirjutatud ning antud operatsioone kasutavad programmid töötavad ka mittelokaalsetel failidel. Mida madalama taseme operatsioonide jaoks see väide kehtib, seda suurem on ligipääsu läbipaistvus. Kui hajusa failisüsteemi sõlmede operatsioonisüsteemiks on UNIX, saavutatakse maksimaalne ligipääsu läbipaistvus siis, kui failitöötluse süsteemifunktsioone saab rakendada ka kaugetele failidele. Süsteemi ressursside *asukoha läbipaistvus* (*location transparency*) tähendab esiteks, et failinimed ei tohi sisaldada viiteid nende füüsilisele asukohale (näiteks serverarvuti nime). Kui see nii ei ole, muutub failide paigutamine ühe sõlme juurest teise juurde üsna tülikaks. Teiseks

peab failinimede ruum[1] olema kaugete failide osas kõigi neid kasutavate protsesside jaoks ühesugune (näiteks klient-server mudeli korral peab suvaline kauge fail asuma erinevates sõlmedes töötavate klientprotsesside jaoks kataloogipuus samal kohal).

Esimeste hajusate failisüsteemide loomise ajal olid olemas vaid efektiivsed kohtvõrgutehnoloogiad, mistõttu süsteem ei saanud oma füüsilistelt mõõtmetelt ja sõlmede arvult kuigi suureks kasvada. Laivõrkude areng viimase aastakümne jooksul on teinud võimalikuks suuremate hajusate failisüsteemide loomise, kus sõlmede arvule ja süsteemi geograafilisele ulatusele enam põhimõttelisi piiranguid ei ole ja süsteem võib kuitahes palju laieneda. Seetõttu peaks tänapäeva hajus failisüsteem olema võimalikult *skaleeritav* (*scalable*). *Skaleeritavuse* (*scalability*) all mõistetakse süsteemi võimet laieneda, tulles samas toime failiteenust kasutavate protsesside poolt genereeritud töökoormuse kasvuga ning jäädes võrdlemisi kergesti hooldatavaks. Et süsteem oleks võimalikult skaleeritav, peab süsteemis olema võimalikult vähe *pudelikaelu* (*bottlenecks*) - süsteemi kasvu takistavaid komponente. See, millised süsteemi komponendid võivad süsteemi laienemisel osutuda pudelikaelteks, sõltub suurel määral mudelist, millele süsteem on üles ehitatud, ja kasutatavast tarkvarast. Klient-server mudelil põhinevas hajusas failisüsteemis on põhilisteks pudelikaelteks serverarvuti ja sellega ühendatud kohtvõrk. Süsteemi pideval laienemisel jõutakse lõpuks hetkeni, kus serverarvuti pole enam piisavalt võimas kõigi klientprotsesside teenindamiseks ning kus serverarvutiga ühendatud kohtvõrk on pidevalt üle koormatud (kuna kõik serverprotsessile saadetud nõuded ja serverprotsessi vastused peavad selle kohtvõrgu läbima).

Skaleeritavusel on palju aspekte [Blaze93]. *Skaleeritavus populatsiooni suhtes* (*population scalability*) tähendab hajusa failisüsteemi võimet taluda selliste sõlmede *arvulist* juurdekasvu, kus töötavad failiteenust kasutavad protsessid. Kui näiteks serverprotsess hoiab iga klientarvuti kohta operatiivmälus hulgaliselt informatsiooni, siis pole süsteem populatsiooni suhtes kuigi skaleeritav, sest serverarvuti operatiivmälu osutub pudelikaelaks. Klientarvutite arvu kasvamisel saabub lõpuks hetk, kus info hoidmiseks ei jätku enam serverarvuti mälu. *Skaleeritavus koormuse suhtes* (*traffic scalability*) tähendab hajusa failisüsteemi (täpsemalt sellesse kuuluvate sõlmede ja

[1] Failinimede ruumi all mõistetakse *absoluutsete failinimede* (*full path names*) ja failide vahelist vastavust [Siegel92].

võrgu) võimet taluda laienemise tagajärjel kasvanud failiteenust kasutavate protsesside poolt genereeritud töökoormust. Klient-server mudelil põhinev hajus failisüsteem pole koormuse suhtes kunagi väga skaleeritav. Skaleeritavus populatsiooni ja koormuse suhtes on põhimõtteliselt erinevad skaleeritavuse aspektid, sest väikese sõlmede arvuga hajusas failisüsteemis töötavad protsessid võivad süsteemile asetada suure koormuse ja vastupidi. *Skaleeritavus administreerimise suhtes* (*administrative scalability*) tähendab süsteemi võimet jääda võrdlemisi kergesti hooldatavaks ka pärast süsteemi laienemist.

Hajusa failisüsteemi skaleeritavust koormuse suhtes suurendab märgatavalt *vahemälude* (*caches*) ning *replikatsiooni* (*replication*) kasutamine. Replikatsioon tähendab, et enamik jagatavaid faile paigutatakse korraga mitme sõlme (klient-server mudeli korral serverarvuti) juurde ning neid haldavad erinevad protsessid (klient-server mudeli korral serverprotsessid), mis võimaldab koormust mitme sõlme vahel ära jagada ja seega vähendada tõenäosust, et mingi sõlm osutub süsteemi laienemist takistavaks pudelikaelaks. Vahemäludest tuleb lähemalt juttu edaspidi.

Efektiivne hajus failisüsteem peab olema võimalikult *veatolerantne* (*fault tolerant*) ehk vigasid taluv. Süsteemi nimetatakse veatolerantseks ehk vigasid taluvaks, kui ta suudab oma töös tekkivaid vigu avastada ning neid kasutajate eest maskeerida või veasituatsioonis oma töö kindlaksmääratud viisil lõpetada [Coul94]. Kõige olulisem veataluvuse aspekt on süsteemi ressursside *kättesaadavus* (*availability*) [Kumar94] - kauged failid peavad võimalike süsteemi töös tekkivate vigade korral jääma neid vajavatele protsessidele kättesaadavaks. Kaugete failide kättesaadavuse suurendamiseks kasutatakse põhiliselt replikatsiooni. Kui üks faili haldav protsess muutub oma töö avariilõpu või võrgurikke tõttu kättesaamatuks, on failiteenust kasutaval protsessil võimalik pöörduda teise faili haldaja poole. Kaugete failide täielikku kättesaadavust pole kunagi võimalik saavutada, sest alati võib leiduda mõni seda takistav viga (näiteks klientarvuti võrguühendused kõigi serverarvutitega katkevad).

Samuti on hajusa failisüsteemi korral oluline selle *julgeolek* (*security*) - tuleb tagada, et informatsiooni saavad omandada või muuta ainult selleks volitatud kasutajad [Sat89]. Andmete omavoliliseks omandamiseks või muutmiseks kasutatakse tavaliselt kommunikatsioonikanalite pealtkuulamist, neid mööda liikuvate teadete omavolilist muutmist või korduvat saatmist ning *matkimist* - pahasoovlikel eesmärkidel käivitatud

protsess saadab teateid ja võtab neid vastu, teeseldes volitatud kasutajale kuuluvat protsessi või faile haldavat protsessi. Hajus failisüsteem peab olema nii kujundatud, et selliste tehnikate rakendamine ei annaks mingeid tulemusi.

Levinuim loetletud rünnakutehnikate vastu kasutatav abinõu on mööda kommunikatsioonikanalit liikuvate teadete kodeerimine erilise võtme abil. Teate dekodeerimine on võimalik teise erilise võtme olemasolul, ilma seda võtit teadmata on teadete sisu võimatu (või väga raske) omandada. Teadete sisu muutmiseks on seega vaja teada mõlemat võtit. *Salajase võtme kodeerimise* (*secret key encryption*) korral kasutatakse kodeerimiseks ja dekodeerimiseks üht ja sama *salajast võtit* (*secret key*). Tuntuim salajase võtme kodeerimise algoritm on DES [Coul94].

Kui kodeerimist kasutatakse kahe osapoole (kasutaja ja faile haldava protsessi) vahelise suhtluse ajal, on põhiprobleemiks see, kuidas vältida kodeerimiseks ja dekodeerimiseks vajalike võtmete sattumist kolmanda osapoole kätte ning kuidas saab üks osapool olla kindel teise *autentsuses* (s.t. et teine osapool on tõesti see, kes ta väidab end olevat). Nende küsimuste lahendamiseks on välja töötatud hulk *autentimisprotokolle* (*authentication protocols*), mille kohased tegevused ehk *autentimine* (*authentication*) viiakse läbi kahe osapoole vahelise suhtluse alguses. Autentimise käigus omandavad mõlemad osapooled (ja ainult nemad) kodeerimise ja dekodeerimise jaoks vajalikud võtmed ning veenduvad ühtlasi teineteise autentsuses. Hilisema suhtluse vältel tõestab osapoole autentsust see, kui temalt saadud teade on korrektselt kodeeritud (selle kindlakstegemiseks võib teadetes sisalduda näiteks kontrollsumma), sest korrektne kodeerimine on võimalik ainult selleks ettenähtud võtme abil.

1.4 Failiteenuse mudel

Selles punktis vaadeldakse failiteenuse mudelit, et tuua sisse edaspidises vajalikud mõisted ning anda ülevaade, kuidas tüüpiline klient-server mudelil põhinev hajus failisüsteem on üles ehitatud.

Käsitletav näidisfailiteenus koosneb kahest eraldiseisvast teenusest - *lamedast failiteenusest* (*flat file service*) ja *kataloogiteenusest* (*directory service*). Failiteenuse sellist komponentideks lahutamist on kasutatud mitmes eksperimentaalses hajusas

failisüsteemis, nagu näiteks Deceit [Siegel92] ja XDFS [Coul94]. Tinglikult võib failiteenuse osaks lugeda veel kliendimooduli - klientarvuti protsessi või klientarvuti operatsioonisüsteemi osa, mis võimaldab klientprotsessidel kasutada lamedat failiteenust ja kataloogiteenust üle arvutivõrgu. Iga klientprotsess kuulub mingile kasutajale, kelle nime all teenuste kasutamine toimub. Lameda failiteenuse ja kataloogiteenuse liideses kirjeldatud operatsioonid on realiseeritud kaugprotseduuridena.

Kaugete failide ja kataloogide integreerimiseks klientarvuti lokaalse failisüsteemiga on kõigi klientarvutite ketastel kokkulepitud nimega kataloog (näiteks `/import`), mis klientprotsesside vaatevinklist sisaldab kõiki kaugeid faile ja katalooge, justkui asuksid need lokaalsel kettal. On garanteeritud, et sellest kataloogist alguse saav kataloogipuu haru on kõigi klientarvutite puhul ühesugune. Sellist lähenemist kasutavad paljud praktilised hajusad failisüsteemid, kuna nii saavutatakse täielik asukoha läbipaistvus.

Lameda failiteenuse liides (vt. Tabel 1.1) võimaldab klientprotsessidel kaugetest failidest lugeda ja neisse kirjutada, lugeda ja seada kaugete failide atribuute (vt. Tabel 1.2) ning kaugeid faile luua ja kustutada. Kauge failiga opereerimiseks tuleb tekstilise nime asemel ette anda unikaalne failiidentifikaator (edaspidi UFID), mis kujutab endast pikka täisarvu.

`Read(File,i,n) → Data`	loeb failist `File` `n` baiti alates `i`-ndast baidist, tagastades need massiivis `Data`.
`Write(File,i,Data)`	kirjutab faili `File` alates `i`-ndast baidist massiivi `Data` sisu.
`Create() → UFID`	loob uue faili, tagastades selle identifikaatori `UFID`.
`Delete(File)`	kustutab faili `File`.
`GetAttributes(File) → Attr`	tagastab struktuuris `Attr` faili `File` atribuudid.
`SetAttributes(File, Attr)`	seab faili `File` atribuudid vastavalt struktuurile `Attr`.

Tabel 1.1 Lameda failiteenuse liides (parameeter `File` tähistab faili UFIDit).

Faili tüüp
Juurdepääsunimekiri (nimekirja iga element koosneb kasutaja või kasutajategrupi nimest ja õigustest antud faili suhtes. Õigusteks võivad olla näiteks lugemis- ja kirjutamisõigus)
Viidete (faili erinevate nimede) **arv**
Faili omanik
Faili suurus
Viimase kasutamise aeg
Viimase modifitseerimise aeg
Atribuutide viimase modifitseerimise aeg

Tabel 1.2 Tüüpilised failiatribuudid.

Kuigi lameda failiteenuse server (ehk *lame failiserver*) haldab iga faili *juurdepääsunimekirja* (*access control list*), ei kontrolli ta klientprotsesside volitusi failide kasutamiseks, sest volitused faili suhtes sõltuvad volitustest kataloogi suhtes, kus fail asub. Lamedal failiserveril ei ole aga mingeid andmeid failide kataloogides paiknemise kohta. Klientprotsessi poolt esitatud korrektne UFID loetakse piisavaks tõendiks, et tal on õigus failioperatsiooni läbi viia.

Võrreldes UNIXi failitöötluse süsteemifunktsioonidega pakub lameda failiteenuse liides samasuguseid võimalusi. Puudub analoog süsteemifunktsioonile `open`, sest faili UFID omandatakse kataloogiteenuse abil. Liideses ei leidu ka süsteemifunktsiooni `close` analoogi, sest erinevalt UNIXist ei eraldata kaugele failile selle igal avamisel uut identifikaatorit, mis pärast faili kasutamist tuleb vabaks kuulutada. Lugemise ja kirjutamise korral antakse erinevalt UNIXist ette failipositsioon, millest alates vastav operatsioon läbi viia (UNIXi puhul määrab positsiooni süsteemi poolt meeles peetud *failiakna* asukoht, mida iga lugemis- ja kirjutamisoperatsiooni järel edasi nihutatakse).

Kõik operatsioonid (välja arvatud `Create`), mida lameda failiserveri vahendusel saab failidele rakendada, on *idempotentsed* (*idempotent*), s.t. nende korduv rakendamine annab sama tulemuse nagu ühekordne rakendamine. Operatsioonide idempotentsus võimaldab nende väljakutsumisel kasutada serverarvuti mälu seisukohalt odavamat vähemalt-üks-kord-semantikat, mille korral serverprotsessis sisalduv kommunikatsioonimoodul ei pea juba tagastatud vastuseid meeles pidama (vt. punkt 1.2).

Kataloogiteenuse liides (vt. Tabel 1.3) võimaldab etteantud tekstilise nime järgi leida sellele vastava faili või kataloogi UFIDi, samuti viia läbi operatsioone failinimede

ruumiga (näiteks faili ümbernimetamine). Selleks on kataloogiteenuse serveri (ehk *kataloogiserveri*) valduses info kaugete failide ja kataloogide paiknemise kohta kataloogipuus. Kataloogiserver haldab iga kataloogi puhul *kataloogifaili*, mis sisaldab kirjeid *<faili või kataloogi nimi, UFID>* antud kataloogis asuvate failide ja kataloogide kohta. Kataloogifail asub lameda failiserveri valduses ning on tavaline fail, millel on samuti UFID ning atribuudid. Kataloogiserver saab kataloogifaile kasutada läbi lameda failiteenuse operatsioonide ning on seega ise lameda failiteenuse klient.

`Lookup(Dir, Name, Access, UserID) → UFID`	leiab kataloogis `Dir` asuva faili või kataloogi tekstilise nime `Name` põhjal `UFID`i. Parameeter `Access` näitab, kuidas parameetriga `UserID` etteantud kasutajale kuuluv klientprotsess kavatseb faili või kataloogi kasutada.
`AddName(Dir, Name, UFID, UserID)`	loob kataloogis `Dir` uue tekstilise nime `Name`, lisades kataloogifaili paari (`Name,UFID`).
`UnName(Dir, Name)`	kõrvaldab kataloogist `Dir` tekstilise nime `Name`.
`ReName(Dir, OldName, NewName)`	muudab kataloogis `Dir` asuva tekstilise nime `OldName` nimeks `NewName`.
`GetNames(Dir, Pattern) → Names`	tagastab kõik kataloogis `Dir` asuvad tekstilised nimed `Names`, mis sobivad regulaarse avaldisega `Pattern`.

Tabel 1.3 Kataloogiteenuse liides (parameeter `Dir` tähistab kataloogi UFIDit).

Kataloogiserveri ülesandeks on ka klientprotsesside volituste kontroll. Kui klientprotsess kavatseb failile või kataloogile mingit operatsiooni rakendada, tuleb selle UFIDi omandamiseks operatsiooni `Lookup` abil kataloogiserveri poole pöörduda, kusjuures faili või kataloogi kasutamise viis antakse ette parameetriga `Access`. Kui omandatud UFIDiga failile või kataloogile kavatsetakse rakendada operatsioone `Read`, `GetAttributes`, `Lookup` või `GetNames`, on parameetri `Access` väärtuseks `READ`, ülejäänud operatsioonide korral (välja arvatud `Create`) on parameetri `Access` väärtuseks `WRITE` (operatsioon `Create` on erand, sest ta ei nõua ainsana faili UFIDi eelnevat omandamist operatsiooni `Lookup` abil). Iga operatsiooni korral kontrollib kataloogiserver faili või kataloogi juurdepääsunimekirja järgi, kas klientprotsessi omanikul on selleks õigust (nimekirja saab ta omandada lameda failiteenuse

operatsiooni `GetAttributes` abil). Kui klientprotsessi volitused on piisavad, saadab kataloogiserver vastuseks faili või kataloogi UFIDi, misjärel pöördutakse kavandatud operatsiooni abil lameda failiserveri või kataloogiserveri poole, andes ette omandatud UFIDi.

Operatsioon `AddName` tekitab etteantud UFIDiga failile või kataloogile uue tekstilise nime ehk *viite* (*link*) ja suurendab viidete arvu. Kui uuele failile või kataloogile omistatakse nimi esimest korda ning viidete arv saab võrdseks ühega, salvestatakse parameetriga `UserID` antud kasutaja faili või kataloogi omanikuna ning juurdepääsunimekiri seatakse vaikimisi väärtusele. Viidete arvu ja juurdepääsunimekirja saab kataloogiserver muuta lameda failiteenuse operatsiooni `SetAttributes` abil. Operatsioon `UnName` kõrvaldab tekstilise nime ning vähendab viidete arvu vastava UFIDiga failile või kataloogile. Kui viidete arv saab võrdseks nulliga, kustutakse fail või kataloog lameda failiteenuse operatsiooni `Delete` abil.

Kuna kataloogid on realiseeritud tavaliste failidena, kasutatakse terminit *fail* erialases kirjanduses (ning edaspidi ka selles töös) mõnikord suvalise kataloogipuus sisalduva objekti tähistamiseks.

Kliendimoodul paikneb füüsiliselt klientarvuti juures. Kliendimoodulis on realiseeritud üldised operatsioonid tööks kaugete failidega (näiteks kaugest failist lugemine, vt. Joonis 1.3), mis sisaldavad omakorda vahetuid pöördumisi kataloogiserveri ja lameda failiserveri poole. Praktiliste hajusate failisüsteemide korral realiseeritakse kliendimoodul kas omaette protsessina või operatsioonisüsteemi tuumas; maksimaalse ligipääsu läbipaistvuse saavutamiseks modifitseeritakse operatsioonisüsteemi tuuma failitöötluseks mõeldud vahendeid, et kauge faili korral kasutataks kliendimoodulis realiseeritud operatsioone. Näiteks UNIXi korral võimaldab selline lähenemine klientprotsessidel pöörduda failitöötluse süsteemifunktsioonide poole, teadmata, kas fail asub lokaalsel kettal või mitte.

```
ReadRemoteFile(FileName, Position, Count, UserID, Data)
{ FileName - kauge fail, kust loetakse andmeid}
{ Position - failipositsioon, millest alates andmeid loetakse}
{ Count - loetavate baitide arv }
{ UserID - kasutaja, kelle nime all lugemine toimub }
{ Data - massiiv, milles tagastatakse loetud baidid }
begin
 file := 0;                  {leida faili UFID}
 repeat                      {ja omistada see muutujale file}
   Part := FindNextPart(FileName);
   file := Lookup(file, Part, READ, UserID);
   if file = -1  then exit(NO_ACCESS);
 until IsLastPart(Part, FileName);
 Data := Read(file, Position, Count); { lugeda failist }
end;
```

Joonis 1.3 Kliendimoodulis realiseeritud operatsioon kaugest failist lugemiseks.

Kaugele failile võidakse viidata mitmeosalise nimega. Sellisel juhul toimub nime interpreteerimine kliendimoodulis - faili UFIDi leidmiseks jagatakse nimi osadeks ning rakendatakse neile järjest operatsiooni `Lookup`. Selle kataloogi UFIDiks, kus asuvad kõik kauged failid ja kataloogid, peab olema mingi kokkulepitud väärtus (näiteks 0). Kui selleks kataloogiks on näiteks `/import`, pöördutakse enne kaugesse faili `/import/home/john/letter.doc` kirjutamist selle faili UFIDi omandamiseks operatsiooni `Lookup` poole kolm korda:

```
Lookup(0, “home”, READ, UserID) → 12
Lookup(12, “john”, READ, UserID) → 512
Lookup(512, “letter.doc”, WRITE, UserID) → 12098
```

Põhimõtteliselt võiks kaugete failide tekstiliste nimede interpreteerimise jätta kataloogiserveri ülesandeks, kuid see tõstab märgatavalt serveri koormust [Howard88]. Seetõttu sellist lähenemist praktilistes hajusates failisüsteemides ei kasutata.

Failiteenuse realiseerimisel on kasulik koondada kauged failid suurematesse rühmadesse ehk *failigruppidesse*. Failigrupp on vähim loogiline ühik, mida lame failiserver võib hallata; ta võib hooldada ka mitut gruppi. Igal failigrupil peab olema identifikaator - uue grupi moodustamisel võib identifikaatori unikaalsuse garanteerimiseks võtta selleks näiteks serverarvuti võrguaadressi, millele lisatakse loomise kellaaega, kuupäeva ja aastat sisaldav täisarv. Faili UFIDi unikaalsuse

tagamiseks grupi piires võib lame failiserver järjekordse faili loomisel enda halduses olevas grupis võtta selle faili *numbriks* täisarvulise loenduri väärtuse ja suurendada seejärel loendurit ühe võrra. UFIDi moodustamiseks lisatakse faili numbrile kogu süsteemis unikaalne failigrupi identifikaator.

Kui lameda failiteenuse servereid on mitu, võib tekkida raskusi failigrupi asukoha (ja seega ka faili asukoha) leidmisega. Iga lame failiserver võib hallata failigrupi asukohtade andmebaasi kirjetega <*failigrupi identifikaator, serverarvuti aadress*>. Kui kliendimoodul on omandanud kataloogiserverilt faili UFIDi, milles sisaldub ka failigrupi identifikaator, on tal kauge faili asukoha kindlakstegemiseks võimalik pöörduda suvalise lameda failiserveri poole.

Failiteenuse kaheks komponendiks lahutamine tekitab mõningaid süsteemi julgeolekuga seotud probleeme. Nimelt eeldab lame failiserver, et faili UFIDi esitamisel on klientprotsessil õigus failile suvalist operatsiooni rakendada; kui operatsioon `Lookup` kutsutakse faili UFIDi omandamiseks välja kasutaja (klientprotsessi omaniku) nime alt ja kasutajal pole faili suhtes vajalikke õigusi, siis kataloogiserver faili UFIDit ei tagasta. Nende eelduste puhul tuleb lahendada järgmised küsimused:

- UFIDid peavad olema võltsimiskindlad, s.t. volitamata kasutajal ei tohiks olla kerge faili UFIDit ise genereerida.
- Tuleb garanteerida, et kasutajad ei ületaks õigusi (näiteks lugemisõiguse põhjal faili UFIDi omandanud kasutaja ei modifitseeriks faili).

Võimalikud lahendused neile probleemidele on toodud allikas [Coul94].

Lameda failiteenuse server pöördub oma töö käigus otse kettablokkide poole abstraktsete operatsioonidega `GetBlock` ja `PutBlock` (vastavalt bloki lugemiseks ja kirjutamiseks). Kuna kettapöördused on reeglina aeglased, ei ole neil operatsioonidel mõtet töötada vahetult kettaga, vaid serverarvuti operatiivmälus asuva *serveri vahemäluga* (*server cache*). Operatsiooni `GetBlock` korral püütakse blokki kõigepealt vahemälust lugeda ja selle puudumisel pöördutakse ketta poole. Kettalt loetud blokk paigutatakse vahemällu, et ta oleks järgmistele lugemisoperatsioonidele kättesaadav. Operatsiooni `PutBlock` korral kirjutatakse modifitseeritud blokk vahemällu ning märgendatakse, et ta tulevikus kettale kirjutataks. Pärast teatud hulka blokioperatsioone võib vahemälu täis saada, misjärel tuleb mõni blokkidest vahemälust välja jätta ja kui ta on märgendatud, siis kettale kirjutada. Väljajäetava bloki valimiseks

on palju meetodeid, mida edaspidi lähemalt kirjeldatakse. Sellise lähenemise korral võivad uuendatud blokid pikaks ajaks vahemällu jääda, ilma et neid kettale kirjutataks. Serverarvuti töö avariilõpu korral lähevad modifitseeritud blokid nii kaduma. Selle vältimiseks kasutatakse tihti *läbikirjutamist* (*write-through*), kus operatsioon `PutBlock` kirjutab peale vahemälu alati ka kettale.

Klientarvuti juures on kaugete failide sisu hoidmiseks otstarbekas kasutada *kliendi vahemälu* (*client cache*), et hoiduda liiga tihedast serverprotsessi poole pöördumisest. Kliendi vahemälu haldab kliendimoodul. Kui serveri vahemälu sisaldab failiblokke, siis kliendi vahemälu elementideks on kas failiblokid või terved failid. Tüüpiliselt paigutatakse element vahemällu siis, kui seda vahemälus ei ole ning klientprotsess sellele viitab, mistõttu elemendi omandamiseks tuleb serverprotsessi poole pöörduda. Kui vahemälu elementideks on *b* baidi suurused blokid, moodustavad iga kauge faili *b* esimest baiti faili esimese bloki, järgmised *b* baiti teise bloki jne.[2] Kui klientprotsess loeb failist süsteemifunktsiooni abil mingi hulga baite alates faili *k*-ndast baidist kuni faili *n*-nda baidini ning neid baite vahemälus ei ole, pöördub kliendimoodul operatsiooni `Read` abil lameda failiserveri poole, kus väärtus *k* on ümardatud alla lähima arvu *b* kordseni ning väärtus *n* on ümardatud üles lähima arvu *b* kordsest ühe võrra väiksema väärtuseni. Kui klientprotsess modifitseerib faili, uuendatakse failile vastavaid vahemälu elemente (faili blokke või faili ennast). Kui uuendamist vajavaid elemente vahemälus pole, loetakse nad eelnevalt vahemällu. Pärast teatud hulka lugemis- või kirjutamisoperatsioone võib vahemälu täis saada, misjärel tuleb mõni element vahemälust välja jätta. Sellise elemendi valikuks on praktilistes hajusates failisüsteemides siiani enim kasutatud LRU-algoritmi, mille kohaselt jäetakse vahemälust välja viimati kõige kauem aega tagasi kasutatud element. Loomulikult tuleb modifitseeritud vahemälu element mingil hetkel lameda failiteenuse operatsiooni `Write` abil serverprotsessile saata. Selle hetke valikuks on palju võimalusi - seda võib teha siis, kui klientprotsess elementi modifitseerib, siis, kui element vahemälust välja jäetakse, jne.

Kui klientarvuti juures asuvas kliendi vahemälus hoitakse elementi (faili või failiblokki), mida teises klientarvutis töötav protsess modifitseerib, siis hetkel, kui

[2] Faili selline blokkideks jaotus on mõtteline ning sel on tähendus vaid kliendi vahemälu jaoks. Serverarvuti ketta blokid, mis antud faili moodustavad, võivad olla hoopis teistsuguse suurusega.

tehtud muudatused serverprotsessini jõuavad, muutub esimese klientarvuti juures asuvas kliendi vahemälus paiknev element *mittekooskõlaliseks* (*inconsistent*) ja tuleb *kehtetuks* (*invalid*) tunnistada. Mittekooskõlaliseks muutunud vahemälu elementide kindlakstegemiseks ehk *vahemälu valideerimiseks* (*cache validation*) kasutatakse mitmesuguseid algoritme. Enamlevinud on kaks vahemälu valideerimise meetodit ja nende modifikatsioonid. *Kliendi juhitud valideerimise* (*client driven validation*) korral pöördub kliendimoodul igal vahemälu elemendile viitamisel serverprotsessi poole, kontrollides viidatud elemendi kooskõlalisust elemendile vastava serverarvuti kettal paikneva faili viimase modifitseerimise aja järgi. *Serveri juhitud valideerimise* (*server driven validation*) korral teatab serverprotsess mingi faili modifitseerimisel kõigile selle faili sisu vahemälus hoidvatele kliendimoodulitele, et failile vastavad vahemälu elemendid on mittekooskõlaliseks muutunud.

Kliendi vahemälud vähendavad serverarvutiga ühendatud kohtvõrgu ning serverarvuti protsessori ja ketta koormust, sest osa failioperatsioonide täitmiseks vajalikke andmeid asub klientarvutite juures. Serveri vahemälud vähendavad serverarvuti ketta ja protsessori koormust, sest iga saabunud nõude rahuldamiseks pole serverprotsessil vaja enam serverarvuti ketta poole pöörduda. Vahemälud võimaldavad suurendada hajusa failisüsteemi skaleeritavust, sest süsteem võib rohkem laieneda, ilma et serverarvuti ja serverarvutiga ühendatud kohtvõrk veel laienemist takistavateks pudelikaelteks muutuksid. Vahemälud võimaldavad samuti kiirendada klientprotsesside tööd, sest klientprotsessile vajalike andmete lugemine kliendi või serveri vahemälust toimub tunduvalt kiiremini kui nende lugemine serverarvuti kettalt. See tähendab, et vahemälud on vajalikud hoolimata sellest, kas süsteem tulevikus laieneb või mitte.

Peale vahemälude kasutatakse hajusa failisüsteemi skaleeritavuse suurendamiseks ka replikatsiooni. Replikatsioon on võte, kus mitu erinevat protsessi pannakse ühtesid ja samu faile haldama. Faili erinevate haldajate käes olevaid faili koopiaid nimetatakse *repliikideks* (*replicas*). Nagu eelpool mainitud, suurendab replikatsiooni kasutamine nii hajusa failisüsteemi skaleeritavust (faili kasutamisel tekkiv koormus jaotub sõlmede vahel, mille juures asuvad repliigid) kui ka andmete kättesaadavust (ühe failihaldaja kättesaamatuks muutumisel võib pöörduda teise haldaja poole).

Siiani on replikatsiooni kasutatud põhiliselt klient-server mudelil põhinevate hajusate failisüsteemide korral, kus ühed ja samad failid asuvad mitmel erineval serverarvutil mitme erineva serverprotsessi valduses. Kui klientprotsess faili modifitseerib, on tehtud muudatuste levitamiseks kaks võimalust. Esimesel juhul pöördub kliendimoodul ainult ühe serverprotsessi poole, kes peab muudatused kõigile ülejäänud repliike haldavatele serverprotsessidele edastama (sellist lähenemist kasutatakse hajusas failisüsteemis Deceit [Siegel92]), teisel juhul pöördub kliendimoodul kõigi repliike haldavate serverprotsesside poole (sellist lähenemist kasutatakse hajusas failisüsteemis Coda [Sat93]).

Suurim probleem, mis replikatsiooni kasutamisel tekib, on võrguvigade tagajärjel tekkiv repliikide mittekooskõlalisus (faili F repliiki nimetatakse *kooskõlaliseks*, kui see kajastab kõiki faili F tehtud muudatusi). Kui näiteks hajusas failisüsteemis Coda ei saa kliendimoodul võrguvea tõttu kõigi serverprotsessidega ühendust võtta, siis faili modifitseerimisel jäävad muudatused mõnedesse repliikidesse sisse viimata. Halvimal juhul võib võrguvea tõttu tekkida olukord, kus mingi faili F korral ei leidu enam ühtegi faili F kooskõlalist repliiki. Näiteks marsruuteri rike võib hajusa failisüsteemi kaheks omavahel ühendamata osaks jagada, mis mõlemad sisaldavad faili F repliike. Kui rikke ajal mõlemas hajusa failisüsteemi osas faili F modifitseeritakse, tekib rikke kadumisel olukord, kus hajusas failisüsteemis pole ühtegi faili F repliiki, mis kajastaks kõiki faili F tehtud muudatusi.

Selle probleemi lahendamiseks on kaks võimalikku meetodit, mille alusel võib replikatsioonistrateegiad jaotada kaheks [Sat93]. *Optimistliku replikatsiooni* (*optimistic replication*) korral lubatakse faile alati modifitseerida, isegi kui see toob kaasa kooskõlaliste repliikide kadumise; samas leiduvad abinõud repliikide kooskõlalisuse taastamiseks. Optimistlikku replikatsiooni on väga põhjalikult käsitletud allikas [Kumar94]. *Pessimistliku replikatsiooni* (*pessimistic replication*) korral püütakse tagada kooskõlaliste repliikide säilimist, keelates teatud juhtudel failide modifitseerimise. Levinuim on võte, kus üks repliikidest kuulutatakse *põhirepliigiks* (*master replica*), mille kooskõlalisus on alati garanteeritud. Selle lähenemise korral saadetakse kõik muudatused alati põhirepliiki haldavale serverprotsessile, kes need harilikke repliike haldavatele serverprotsessidele edastab. Kui põhirepliigi haldaja on kättesaamatu, pole faili modifitseerimine võimalik. Küllalt levinud on ka võte, kus kooskõlaliste repliikide säilitamiseks nõutakse faili modifitseerimisel *kvoorumi* (*quorum*) olemasolu. Kui

kvoorum on Q repliiki, siis on faili modifitseerimine lubatud vaid juhul, kui on tagatud muudatuste levimine vähemalt Q repliigini. Sobivalt valitud kvoorum võimaldab säilitada faili kooskõlalisi repliike neil juhtudel, kus hajus failisüsteem jaguneb võrguvea tõttu kaheks üksteisest lahutatud osaks ning mõlema osa puhul saavad suvalised kaks sellesse ossa kuuluvat sõlme üksteisega ühendust pidada. Kui hajusas failisüsteemis on faili F repliike kokku R tükki ning $2Q > R$, on garanteeritud, et mõned faili F repliigid jäävad kooskõlaliseks. Kvoorumeid kasutatakse näiteks hajusas failisüsteemis Deceit [Siegel92].

2. Näiteid hajusate failisüsteemide kohta

Töö selles osas vaadeldakse kahte maailmas enamlevinud klient-server mudelil põhinevat hajusat failisüsteemi, mis on loodud erinevate eesmärkide saavutamiseks. NFSi autorite eesmärgiks oli luua mõnekümnest sõlmest koosneva hajusa failisüsteemi tarkvara. AFSi autorid pidasid seevastu kõige olulisemaks hajusa failisüsteemi skaleeritavust, et süsteem võiks koosneda paljudest sõlmedest ja vajadusel ühendada endas mitme organisatsiooni arvuteid ning kohtvõrke.

2.1 NFS

1980-ndate aastate alguseks oli loodud hulk eksperimentaalseid hajusaid failisüsteeme, mida kasutati edukalt mitmetes uurimislaboratooriumides ja ülikoolides. Neist ükski ei olnud aga kuigi levinud - süsteemid olid loodud konkreetse organisatsiooni tarbeks, kasutajate arv jäi harilikult saja piiridesse.

1984. aastal alustati firmas Sun Microsystems töid uut tüüpi hajusa failisüsteemi NFS (Network Filesystem) loomiseks, mis ühendaks endas eelmiste süsteemide kogemused ja oleks universaalne - kasutatav nii akadeemilises kui kommertsmaailmas. Esimest korda esitleti NFSi laiemale avalikkusele 1985. aasta USENIXi Suvekonverentsil ning konverentsi käsitlevas kogumikus avaldati süsteemi autorikirjeldus [Sand85]. Nüüdseks on NFS kõige enam levinud hajus failisüsteem maailmas ning teda loetakse tarkvaratööstuslikuks standardiks - peaaegu kõik erinevate tarkvaratootjate poolt loodud operatsioonisüsteemid sisaldavad NFSi kasutamise tuge. Ta on oluliselt mõjutanud ka teisi hiljem loodud hajusaid failisüsteeme.

NFSi tarkvara kirjutati algselt UNIX-operatsioonisüsteemiga arvutite jaoks. Kuigi nüüdseks on olemas versioonid ka teiste operatsioonisüsteemide jaoks, on mitmed NFSi failiteenuse operatsioonid UNIXi-laadsed (näiteks viidete tekitamine). Selles punktis käsitletaksegi NFSi versiooni, mis on mõeldud UNIXi failisüsteemi emuleerimiseks.

2.1.1 Kliendimoodul ja servermoodul

NFSi kliendimoodul on realiseeritud klientarvuti operatsioonisüsteemi tuumas. Maksimaalse ligipääsu läbipaistvuse saavutamiseks saab kaugeid faile kasutada UNIXi failitöötluse süsteemifunktsioonide abil. Lame failiteenus ja kataloogiteenus on integreeritud ühtseks failiteenuseks. Klientprotsesside kiirema teenindamise huvides pakub serverprotsessi asemel failiteenust serverarvuti operatsioonisüsteemi tuumas realiseeritud *servermoodul* (*server module*). Operatsioonid, mida saab NFSi servermooduli vahendusel kaugetele failidele rakendada, on realiseeritud kaugprotseduuridena Sun RPC nimelise vahenditekomplekti abil, mis on mõeldud kasutamiseks koos C-keele ning põhiliselt TCP ja UDP transpordiprotokollidega [Sun88].

NFSi autorid pidasid väga oluliseks, et servermooduli töö avariilõpu korral toimuks selle taaskäivitamine väga lihtsalt [Sand85]. Selle eesmärgi saavutamiseks kujundati servermoodul *olekuvabaks* (*stateless*) - ta ei pea meeles mingit informatsiooni klientprotsesside oleku kohta (nagu näiteks failiakende positsioonid klientprotsesside poolt avatud failides). Sellisel juhul ei kaasne servermooduli taaskäivitamisega mingeid erilisi info taastamise protseduure. Kui servermoodul ei ole olekuvaba ja hoiab klientprotsesside kohta käivat infot näiteks operatiivmälus, kust see serverarvuti töö avariilõpu puhul kaduma läheb, on servermooduli taaskäivitamine üsna keerukas. Igas klientarvutis asuv kliendimoodul peab servermooduli töö avariilõpu kindlaks tegema ja servermooduli taaskäivitumisel saatma sellele oleku, milleni antud sõlmes töötavad klientprotsessid avariilõpu hetkeks olid jõudnud. Samas on kliendimoodulitel võrdlemisi raske avariilõppu kindlaks teha. Kui kliendimoodulini ei jõua vastus mingi operatsiooni täitmise kohta, siis ei pruugi see tähendada servermooduli töö avariilõppu, vaid võib olla tingitud ka võrgurikkest.

2.1.2 Servermooduli liides

Olulisemad NFSi servermooduli operatsioonid on toodud Tabel 2.1-s [Coul94].

`lookup(dirfh, name) → fh, attr`	Tagastab kataloogis `dirfh` asuvale nimele `name` vastava faili identifikaatori `fh` ja atribuudid `attr`.
`create(dirfh, name, attr) → newfh, attr`	Loob kataloogis `dirfh` faili nimega `name` ja atribuutidega `attr`, tagastades loodud faili identifikaatori `newfh` ja atribuudid `attr`.
`remove(dirfh, name) → status`	Kustutab kataloogis `dirfh` asuva faili `name`, tagastades operatsiooni tulemuse `status`.
`getattr(fh) → attr`	Tagastab faili `fh` atribuudid `attr`.
`setattr(fh, attr) → attr`	Seab faili `fh` atribuudid vastavalt parameetrile `attr` ja tagastab need samas.
`read(fh, offset, count) → attr, data`	Loeb failist `fh` alates positsioonist `offset` `count` baiti, tagastades loetud andmed `data` ning faili atribuudid `attr`.
`write(fh, offset, count, data) → attr`	Kirjutab faili `fh` alates positsioonist `offset` massiivi `data` `count` esimest baiti, tagastades faili atribuudid `attr`.
`rename(dirfh, name, todirfh, toname) → status`	Paigutab kataloogis `dirfh` asuva faili `name` uue nime `toname` all kataloogi `todirfh`, tagastades operatsiooni tulemuse `status`.
`link(newdirfh, newname, dirfh, name) → status`	Loob kataloogis `newdirfh` viite `newname` kataloogis `dirfh` asuvale failile `name`, tagastades operatsiooni tulemuse `status`.
`symlink(newdirfh, newname, string) → status`	Loob kataloogis `newdirfh` sümbolviite `newname`, mille sisuks on `string`, ning tagastab operatsiooni tulemuse `status`.
`readlink(fh) → string`	Tagastab sümbolviite `fh` sisu `string`.
`mkdir(dirfh, name, attr) → newfh, attr`	Loob kataloogis `dirfh` uue kataloogi nimega `name` ja atribuutidega `attr`, tagastades loodud kataloogi identifikaatori `newfh` ja atribuudid `attr`.
`rmdir(dirfh, name) → status`	Kustutab kataloogis `dirfh` asuva kataloogi `name`, tagastades operatsiooni tulemuse `status`.
`readdir(dirfh, cookie, count) → entries`	Loeb kataloogifailist `dirfh` `count` esimest baiti, tagastades loetud kirjed `entries`. Iga tagastatud kirje sisaldab faili nime, selle identifikaatorit ja viita järgmisele kirjele. Parameeter `cookie` on viit mingile kirjele ja näitab, et kataloogifailist tuleb lugeda alates antud kirjest. Kui `cookie` = 0, siis tagastatakse `count` baidi ulatuses kirjeid alates kataloogifaili algusest.

Tabel 2.1 NFSi servermooduli olulisemad operatsioonid (`fh`, `dirfh`, `newdirfh` ja `todirfh` on faili UFIDid ehk *identifikaatorid* ning `name`, `newname` ja `toname` failide nimed; `attr` tähistab faili atribuute sisaldavat struktuuri ja `status` operatsiooni tulemust näitavat koodi).

Operatsioonid `read`, `write`, `getattr`, `setattr` ja `lookup` on analoogilised punktis 1.4 käsitletud lameda failiserveri ja kataloogiserveri vastavate operatsioonidega.

Operatsioonid `read`, `write`, `setattr` ja `lookup` tagastavad kõrvalefektina parameetriga `fh` või `name` antud faili atribuudid. Kuna lame failiteenus ja kataloogiteenus on NFSi korral integreeritud, toimub faili füüsiline loomine ning sellele nime andmine ühe operatsiooniga `create` (kõrvalefektina tagastatakse ka loodud faili atribuudid). Fail kustutatakse samal põhjusel samuti üheainsa operatsiooniga `remove`. Viidete loomine toimub operatsiooniga `link`.

Sümbolviidete (*symbolic links*) tekitamiseks on olemas operatsioon `symlink`. Sümbolviidet kasutatakse mingile failile osutamiseks ja ta kujutab endast erilist tüüpi faili, milles asub osutatava faili nimi. Sümbolviite järgi osutatava faili identifikaatori leidmiseks peab kliendimoodul lugema sümbolviite sisuks oleva nime operatsiooni `readlink` abil, jaotama nime osadeks ning rakendama igale osale operatsiooni `lookup`. Kas fail on sümbolviide või mitte, saab teada operatsiooni `lookup` poolt tagastatud faili atribuutide põhjal (faili tüüp on üks atribuutidest).

Operatsioonide `mkdir` ja `rmdir` abil saab kaugeid kataloge luua ja kustutada. Kataloogi loomisel tagastatakse kõrvalefektina loodud kataloogi atribuudid. Operatsioon `readdir` tagastab etteantud kataloogifailis sisalduvad kirjed. Kui *kirjete arv * kirje suurus baitides* > `count`, saadab servermoodul osalise nimekirja. Igas tagastatud kirjes on viit järgmisele kirjele. Parameetriga `cookie` edastatakse viit sellele kirjele, millest alates tuleb kataloogifailist lugeda. Kui parameetriks `cookie` on eelmisel pöördumisel tagastatud osalise nimekirja viimasest kirjest omandatud viit, saadab servermoodul järgmise osa nimekirjast. Kui parameetri `cookie` väärtuseks on 0, loetakse ja tagastatakse kirjeid alates kataloogifaili algusest.

Suur osa operatsioonidest, mida servermooduli vahendusel saab kaugetele failidele rakendada, on idempotentsed. See võimaldab vähemalt-üks-kord-semantika kasutamist, mistõttu servermoodulis kaob vajadus ajaloomehhanismi (vt. punkt 1.2) rakendamise järele. Operatsioonide idempotentsus on oluline, sest ajaloomehhanismi abil hoitakse meeles infot klientprotsesside kohta ja selle rakendamisel mindaks vastuollu servermooduli olekuvabaduse põhimõttega. Alates NFSi 3. versioonist siiski kasutatakse mitteidempotentsete operatsioonide korral ajaloomehhanismi - pärast selliste operatsioonide täitmist peetakse vastuseid *võimaluse korral lühemat aega* meeles [Call95]. Samas ei ole servermoodulil mingit *kohustust* seda teha ja

kliendimoodul peab sellega arvestama. Servermooduli avariilõpujärgsel käivitamisel ei tehta mingit katset sedalaadi info taastamiseks.

2.1.3 UNIXi failisüsteemi emuleerimine

UNIX lubab kustutada avatud faile, kusjuures pärast kustutamist ja enne faili sulgemist on võimalik faili kirjutada ja sealt lugeda. Paljud UNIXi programmid kasutavad seda võimalust ajutiste failide loomiseks. Samuti võib muuta avatud faili kaitsekoodi nii, et failist pole lubatud lugeda (või sinna kirjutada), kuid UNIX lubab siiski sellest failist lugeda (või sinna kirjutada) niikaua, kuni fail on avatud.

NFSi servermoodul ei suuda sellist failioperatsioonide semantikat ehk *failisemantikat* tagada, sest servermoodulil puuduvad olekuvabaduse tagamiseks faili avamise ja sulgemise operatsioonid ning servermoodul ei hoia mälus infot selle kohta, millised klientprotsessid milliseid faile avatuna hoiavad. Kui klientprotsess kustutab faili või muudab selle kaitsekoodi, ei suuda servermoodul seega otsustada, kas fail on avatud ja millistele klientprotsessidele peaks võimaldatama failiga edasi töötada. Seetõttu on kliendimooduli üheks ülesandeks garanteerida klientprotsessidele see osa UNIXi failisemantikast, mida servermoodul ei suuda toetada [Sand85].

Kui klientprotsess kustutab faili, kontrollib kliendimoodul, kas mõni samas sõlmes töötav protsess hoiab faili avatuna. Kui see nii on, pöördub kliendimoodul kõigepealt operatsiooni `rename` poole ja annab failile teise nime. Failist lugemist ja sinna kirjutamist see ei mõjuta, sest operatsioonide `read` ja `write` parameetriks on faili identifikaator, mis jääb samaks. Samas kaob faili nimi failinimede ruumist, justkui oleks fail kustutatud. Kui ükski protsess enam faili avatuna ei hoia, kustutab kliendimoodul selle operatsiooni `remove` abil [Sand85]. Selline lähenemine tagab UNIXi failisemantika ainult ühes sõlmes töötavate klientprotsesside jaoks, sest kliendimoodulil pole võimalik kindlaks teha, kas kustutatavat faili hoitakse veel avatuna ka mõnes teises sõlmes.

Avatud faili kaitsekoodi muutumisel ei ole kahjuks mingit üldist võimalust selle failiga töötavale klientprotsessile edasise juurdepääsu garanteerimiseks. Servermoodul tagab ainult, et faili omanikule kuuluvatel klientprotsessidel on nii failist lugemine kui sinna kirjutamine *alati* lubatud faili kaitsekoodist olenemata [Sun89].

Kui faili kaitsekoodis on kasutaja jaoks seatud ainult x-bitt, peab NFSi servermoodul siiski lubama kliendimoodulil failist lugeda, sest muidu ei saaks kasutajale kuuluv klientprotsess faili täitmiseks klientarvuti mällu laadida. Sellisel juhul tagab kliendimoodul, et klientprotsess ei saa failist lugeda, kuid saab seda samas täita kui programmi [Sun89].

2.1.4 Virtuaalne failisüsteem ja haakimine

Kui serverarvutite operatsioonisüsteemiks on UNIX, siis on serverarvuti ketta partitsioonile vastav UNIXi failisüsteem vähim loogiline ühik, mida servermoodulid võivad hallata. UNIXi failisüsteemi puhul kirjeldab selle iga faili samas failisüsteemis asuv eriline kirje ehk *i-sõlm* (*i-node*), mis sisaldab vastava faili kohta täielikku informatsiooni (faili atribuudid, millised kettapartitsiooni blokid antud faili moodustavad jms.). Igal i-sõlmel on unikaalne identifikaator ehk *i-number* (*i-number*), mis määrab üheselt i-sõlme füüsilise asukoha failisüsteemis. Seetõttu võib i-numbreid kasutada failide identifitseerimiseks.

Kataloog on UNIXi failisüsteemis realiseeritud tavalise failina, millel on oma i-number ning mis sisaldab kirjeid <*faili nimi*, *i-number*> kõigi antud kataloogis asuvate failide kohta. Failisüsteemi juurkataloogi i-numbriks on mingi kokkulepitud väärtus. See võimaldab faili nime järgi iteratiivselt faili i-numbri leida, mille põhjal saab omakorda leida i-sõlme. Kuna faili i-sõlm kirjeldab faili täielikult, kasutavad i-sõlmes sisalduvat infot UNIXi failitöötlusega seotud süsteemifunktsioonid. Faili avamisel loeb operatsioonisüsteem vastava i-sõlme operatiivmällu, et faili kirjeldavale infole oleks võimalik kiiremini ligi pääseda.

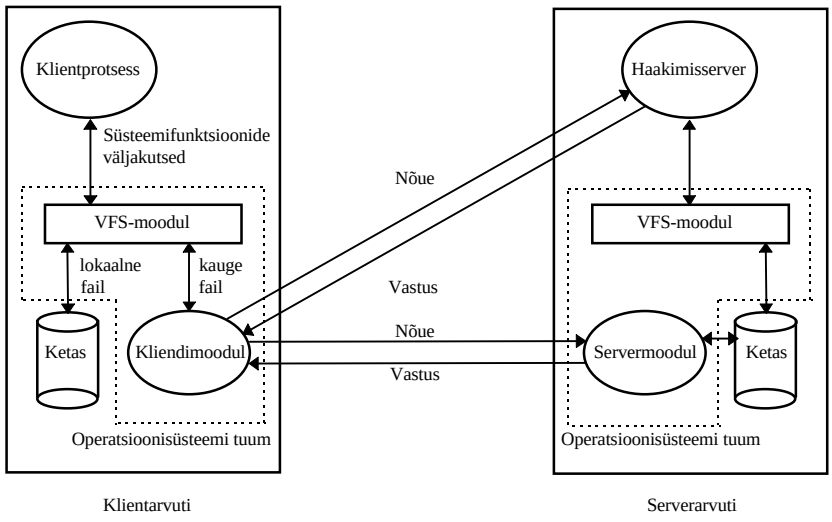
NFSi installatsiooni kuuluvate arvutite operatsioonisüsteemide tuumadele on lisatud eriline *virtuaalse failisüsteemi moodul* ehk *VFS-moodul* (*virtual file system module*), mis võimaldab erinevat tüüpi lokaalsetesse ja kaugetesse (serverarvuti kettal asuvatesse) failisüsteemidesse kuuluvate avatud failide korral kasutada ühesuguseid i-sõlmi, mida nimetatakse *v-sõlmedeks* (*v-nodes*). Sellega võimaldab VFS-moodul muuta süsteemifunktsioonid failisüsteemi tüübist sõltumatuteks (vt. Joonis 2.1) [Kleim86].

Kui failile viidatakse, luuakse operatiivmällu teda täielikult kirjeldav v-sõlm. V-sõlm sisaldab failisüsteemi tüübist sõltumatut infot (nagu näiteks viidete arv), viita failisüsteemi tüübist sõltuvat infot sisaldavale struktuurile (lokaalsesse UNIXi failisüsteemi kuuluva faili korral on selleks faili i-sõlm, kauge faili korral faili identifikaator) ning funktsiooniviitu üldistele failioperatsioonidele, nagu failist lugemine, faili kirjutamine jt., mille sisu on failisüsteemi tüübist sõltuv. Süsteemifunktsioonide väljakutsumisel pöördutakse v-sõlmes sisalduvate funktsiooniviitade kaudu vastavate juba failisüsteemi tüübist sõltuvate operatsioonide poole. Näiteks süsteemifunktsiooni `stat` failile rakendamisel pöördutakse operatsiooni poole, millele osutab faili v-sõlmes sisalduv funktsiooniviit `vn_getattr`. Kui fail on kauge, osutab viit `vn_getattr` kliendimoodulis realiseeritud operatsioonile kauge faili atribuutide lugemiseks, kust pöördutakse omakorda operatsiooni `getattr` abil vahetult servermooduli poole.

Kui fail on kauge, sisaldab v-sõlm viita faili identifikaatorile, mis koosneb failisüsteemi unikaalsest identifikaatorist ja faili i-numbrist. Igale failisüsteemile omistatakse identifikaator tema loomise hetkel. Koos faili i-numbriga määrab ta faili üheselt kogu hajusas failisüsteemis. Probleem on selles, et UNIXi puhul omistatakse kustutatud failide i-numbrid uute loodud failide i-sõlmedele. Nii võib tekkida olukord, kus juba kustutatud faili identifikaatoriga pääseb ligi uuele failile. Selle vältimiseks sisaldab i-number seerianumbrit, mida suurendatakse ühe võrra iga kord, kui i-number uuesti kasutusele võetakse [Sand85].

Et klientprotsessid saaksid mingit serverarvuti kettal asuvat failisüsteemi kasutada, on kliendimoodulil tarvis teada selle failisüsteemi juure identifikaatorit (muidu pole võimalik operatsiooni `lookup` abil iteratiivselt failinimede järgi identifikaatoreid leida). UNIXi puhul muutub mingi lokaalne failisüsteem kättesaadavaks alles pärast kettapartitsioonile vastava seadmefaili *haakimist* (*mounting*) mingisse kataloogi ehk *haakepunkti* (*mount point*), kusjuures haagitud failisüsteemi juur varjab haakepunkti tegeliku sisu. Haakimisel tehakse sissekanne *haaketabelisse* (*mount table*), mis sisaldab kirjeid <*haakepunkt*, *failisüsteem*> tehtud *haagete* (*mounts*) kohta [Silb94]. NFSi autorid pidasid loomulikuks defineerida haakimine lisaks seadmefailidele ka kaugete failisüsteemide kataloogidel, mille käigus kliendimoodul omandab kauge kataloogi identifikaatori.

NFSis võimaldab kaugete kataloogide haakimist eraldi teenus, mille server töötab servermooduliga samas arvutis (vt. Joonis 2.1). *Haakimisteenus* (*mount service*) on realiseeritud omaette teenusena, sest haakimisserver ei ole oma olemuselt olekuvaba - ta haldab kliendimoodulite nimekirja, kes on serverarvuti katalooge haakinud. See võimaldab haakimisserveril servermooduli töö lõppemisel kliendimoodulitele teatada, et kauged kataloogid pole enam kättesaadavad [Sun89]. Haakimisserveri operatsioonid `mnt` ja `umnt` võimaldavad kaugeid katalooge haakida ning lahti haakida, operatsiooni `export` abil saab teada failisüsteemide nimekirja, mille kataloogide haakimine on lubatud (tavaliselt on need failisüsteemid ära toodud serverarvuti failis `/etc/exports`).



Joonis 2.1 NFS.

VFS-moodul võimaldab haagitud lokaalsete failisüsteemide ja kaugete failisüsteemide kataloogide kirjeldamiseks kasutada ühesuguseid failisüsteemi tüübist sõltumatuid *VFS-struktuure* [Kleim86]. VFS-struktuur sisaldab tüübist sõltumatuid failisüsteemi parameetreid (nagu näiteks failibloki suurus), viita tüübist sõltuvaid parameetreid sisaldavale struktuurile (lokaalse UNIXi failisüsteemi korral on selleks struktuuriks haaketabeli kirje) ning funktsiooniviitu üldistele failisüsteemioperatsioonidele nagu haakimine, lahtihaakimine jms., mille sisu sõltub failisüsteemi tüübist.

Maksimaalse ligipääsu läbipaistvuse saavutamiseks modifitseerisid NFSi autorid süsteemifunktsiooni `mount`, et muuta see failisüsteemi tüübist sõltumatuks.

Süsteemifunktsiooni `mount` väljakutsumisel on selle parameetrite hulgas haakepunkt, failisüsteemi tüüp, mille põhjal luuakse klientarvuti operatiivmällu VFS-struktuur, ning tüübist sõltuvad andmed (tüübi "`nfs`" korral serverarvuti aadress ja kauge failisüsteemi kataloog). Haake sooritamiseks kutsutakse süsteemifunktsioonist `mount` VFS-struktuuris sisalduva viida `vfs_mount` kaudu välja failisüsteemi tüübist sõltuv haakimisoperatsioon. Kui failisüsteemi tüübiks on "`nfs`", pöördutakse vastavast operatsioonist omakorda haakimisserveri operatsiooni `mnt` poole, mis tagastab haagitud kataloogi identifikaatori.

NFSi korral haagitakse kauged kataloogid nagu lokaalsed failisüsteemidki tavaliselt klientarvuti käivitamise ajal. Põhimõtteliselt võivad kaks kliendimoodulit ühe ja sama kauge kataloogi erinevatesse lokaalsetesse kataloogidesse haakida. Seetõttu ei garanteeri NFS asukoha läbipaistvust täielikult - failinimede ruum ei ole kaugete failide osas kõigi klientprotsesside jaoks ühesugune, ühed ja samad kauged failid võivad erinevate klientarvutite puhul asuda erinevates kataloogides.

NFSi puhul kasutatakse haakimisel *mittetsentraliseeritud* meetodit [Sat93] - kogu haakimist puudutav info asub klientarvutite juures ning haakimisserver ei kirjuta ette, kuhu serverarvuti kataloogid haakida tuleb. See tähendab, et NFS ei ole administreerimise suhtes eriti skaleeritav. Pärast kauge failisüsteemi paigutamist ühelt serverarvutilt teisele tuleb selle failisüsteemi kataloogid kõigil klientarvutitel uuesti haakida, mistõttu administreerimise maht kasvab enam-vähem proportsionaalselt süsteemi sõlmede arvu kasvades, sest enamik sõlmi on klientarvutid. Selle probleemi lahendamiseks on võetud kasutusele *automaathaakija* (*automounter*) [Coul94] - klientarvuti protsess, millele antakse iga kauge failisüsteemi korral ette serverarvutite nimekiri, mille ketastel see failisüsteem võib asuda. Automaathaakija haagib kauge kataloogi lahti, kui sellele pole teatud aja (tavaliselt 5 minuti) jooksul viidatud. Vajadusel haagib ta kauge kataloogi uuesti sellelt serverarvutilt, mille juures töötav haakimisserver vastab esimesena positiivselt saadetud haakimisnõudele (selline serverarvuti on suure tõenäosusega kõige vähem koormatud). Automaathaakija lahendab probleemi siiski osaliselt, sest kauge failisüsteemi paigutamine nimekirjast puuduvale serverarvutile nõuab ikkagi administratiivset sekkumist klientarvutitel. Samas võimaldab automaathaakija primitiivset laadi replikatsiooni kasutamist, kus

repliikide rolli sobivad harva modifitseeritavad ja korraga mitmel serverarvutil asuvad failisüsteemid.

2.1.5 Julgeolek

Kasutaja ja servermooduli vahelist autentimist võimaldavad läbi viia Sun RPC enda vahendid [Sun88]. Kuna kaugprotseduure kutsutakse välja kliendimoodulist, esindab kasutajaid autentimisel kliendimoodul, viies nende eest läbi kõik vajalikud tegevused. NFSi puhul[3] on kõige levinum DES-autentimisprotokoll, mis on oma nime saanud protokollis kasutatava kodeerimisalgoritmi nime järgi. Kasutaja jaoks toimub autentimine talle märkamatult siis, kui antud kasutaja jaoks kutsutakse kliendimoodulist esimest korda välja mingi kaugprotseduur.

On eeldatud, et kasutajal C on *salajane võti* (*secret key*) SK_C ja *avalik võti* (*public key*) PK_C ning servermoodulil S on salajane võti SK_S ja avalik võti PK_S. Salajast võtit teab ainult selle omanik, avalik võti on seevastu kõigile mingi teenuse (tavaliselt YP või NIS+) vahendusel kättesaadav. Avalik võti PK leitakse salajase võtme SK järgi seosest $PK = B^{SK} \bmod M$, kus konstandid B ja M valitakse nii, et SK leidmine PK põhjal oleks arvutuslikult väga keeruline (üks sobivatest valikutest on toodud allikas [Sun88]).

Enne esimest kaugprotseduuri väljakutset genereerib kliendimoodul kasutaja jaoks salajase DES-algoritmi võtme CK, mida edaspidi kasutajale kuuluvate klientprotsesside ja servermooduli vahelisel suhtlemisel info kodeerimiseks tarvitatakse. Et kasutajat esindav kliendimoodul saaks võtme CK servermoodulile edasi anda, arvutab kliendimoodul veel teise võtme $K = PK_S^{SKc} \bmod M$, millega võti CK enne serverprotsessile saatmist kodeeritakse. Edaspidi tähistab $E(K_e, X)$ tulemust, mis saadakse andmete X kodeerimisel võtmega K_e, ning $D(K_d, Y)$ tulemust, mis saadakse kodeeritud andmete Y dekodeerimisel võtmega K_d. Nõue, mis saadetakse servermoodulile esimesel kaugprotseduuri väljakutsel, sisaldab peale kaugprotseduuri sisendparameetrite veel viit välja (vt. Tabel 2.2).

[3] Sun RPC töötati välja mitte spetsiaalselt NFSi jaoks, vaid suvalises hajussüsteemis kasutamiseks.

kliendi võrgunimi
E(K, CK) - võti CK, mis on kodeeritud võtmega K
E(CK, T) - nõude saatmise aeg T (esitatud täisarvuna), mis on kodeeritud võtmega CK
E(CK, W) - akna suurus W, mis on kodeeritud võtmega CK
E(CK, W-1) - akna suurusest W ühe võrra väiksem arv, mis on kodeeritud võtmega CK
kaugprotseduuri sisendparameetrid

Tabel 2.2 Esmakordne servermoodulile saadetav nõue.

Kliendi võrgunimi (*client network name*) koosneb klientarvuti operatsioonisüsteemi tüübist, kasutaja identifikaatorist selle operatsioonisüsteemi all ja võrgudoomeni nimest, kus asub klientarvuti ja mille piires kasutaja identifikaator on unikaalne. Kuna doomeni nimi on samuti unikaalne, on seda ka võrgunimi. Kliendi võrgunimeks võib olla näiteks `unix.515@sun.com`.

Pärast esimese nõude vastuvõtmist omandab servermoodul S mingilt teenuselt (tavaliselt YP või NIS+) kliendi võrgunime järgi kasutaja avaliku võtme PK_C ja arvutab võtme $K' = PK_C^{SK_S} \bmod M$. Konstandid B ja M on nii valitud, et alati K’ = K. Võtme K’ abil on dekodeerides võimalik leida võti CK’ = D(K’, E(K, CK)). Kuna K’ = K, siis ka CK’ = CK. Võtme CK’ abil saab servermoodul leida nõude saatmise aja T’ = D(CK’, E(CK, T)) ja akna suuruse W’ = D(CK’, E(CK, W)), kusjuures T’ = T ja W’ = W. Volitamata osapoolel pole mingit võimalust võtme CK teadasaamiseks, sest ta ei suuda välja E(K, CK) dekodeerida (see nõuaks võtme SK_S või SK_C teadmist).

Kui volitamata osapool püüab esineda kasutaja C nimel ja saadab servermoodulile vastava kliendi võrgunimega nõude, siis ei ole volitamata osapoolel võimalik võtit CK korrektselt kodeerida (see nõuaks võtme SK_S või SK_C teadmist). Sellisel juhul K’≠K ja seega CK’≠CK. Kui volitamata osapool püüab kinni mööda kommunikatsioonikanaleid liikuva nõude ja modifitseerib kliendi võrgunime, siis kasutab servermoodul võtme K’ arvutamisel vale avalikku võtit PK_C ning jällegi K’≠K ja CK’≠CK. Kui volitamata osapool modifitseerib juhuslikult välja E(K, CK) bitte, siis selle dekodeerimine ebaõnnestub ja CK’≠CK. Kui CK’≠CK, siis

$$D(CK', E(CK, W)) - 1 \neq D(CK', E(CK, W-1)). \qquad (*)$$

Sama juhtub siis, kui volitamata osapool on modifitseerinud välja E(CK, W) ja/või välja E(CK, W-1) bitte. Kui modifitseeritakse välja E(CK, T) bitte, pole T’ = D(CK’, E(CK, T)) nõude saatmise ajana realistlik.

Kui tingimus (*) on täidetud, T’ on nõude saatmise ajana ebareaalne või nõude vastuvõtmise aeg on hilisem kui T’ + W’, heidetakse nõue kõrvale ning vastavat kaugprotseduuri ei täideta. Vastasel korral loeb servermoodul kasutaja autentsuse tõestatuks ning seab mingi teenuse (tavaliselt YP või NIS+) abil kliendi võrgunimele vastavusse kasutaja- ja grupiidentifikaatori, mille all nõutud operatsioon läbi viiakse [Taylor86]. NFSi puhul sisaldavad kaugete failide juurdepääsunimekirjad UNIXi-päraselt 3 elementi: õigused faili omaniku, grupi ja ülejäänud kasutajate jaoks, kusjuures võimalikeks õigusteks on lugemis-, kirjutamis- ning täitmisõigus. Kliendi võrgunime põhjal leitud kasutaja- ja grupiidentifikaatori järgi saab servermoodul kontrollida kasutaja volitusi operatsioonide läbiviimiseks.

Kliendi võrgunimi, DES-algoritmi võti CK’ ning akna suurus W’ salvestatakse servermooduli sisemisse tabelisse. Servermooduli vastus esmakordsele nõudele sisaldab peale kaugprotseduuri väljundparameetrite veel nõude saatmise ajast ühe võrra väiksemat väärtust, mis on kodeeritud võtmega CK’ (s.o. E(CK’, T’-1)), ning sisemise tabeli kasutajale vastava rea numbrit (järgnevatel kaugprotseduuride väljakutsetel antud kasutaja nime alt annab kliendimoodul võrgunime asemel ette omandatud reanumbri). Kuna võti CK’ omandatakse kasutaja nõudest, kus ta on kodeeritud võtmega K, ning keegi peale servermooduli ei suuda võtit K’ (= K) leida, siis pole kellelgi peale servermooduli võimalik sellist vastust kliendimoodulile saata. Ühe lahutamine väärtusest T’ on vajalik, sest volitamata osapool saaks välja E(CK’, T’) (= E(CK, T)) kasutaja enda nõudest kopeerida. Nii saab ka kliendimoodul kui kasutaja esindaja servermooduli autentsuses veenduda.

Kuna DES-autentimisprotokolli käigus omandavad mõlemad osapooled ja ainult nemad DES-algoritmi võtme CK, saaks seda kasutada kaugprotseduuride sisend- ja väljundparameetrite kodeerimiseks. Seda aga siiski ei tehta, kuna Sun RPC väljatöötajad ei pidanud võrgu pealtkuulamise probleemi oluliseks [Taylor86]. Pärast esimest kaugprotseduuri väljakutset ja edukat autentimist saadab kliendimoodul selle asemel järgnevates nõuetes lisaks kaugprotseduuri sisendparameetritele veel nõude saatmise aja T võtmega CK kodeeritult (s.o. E(CK, T)) ning servermoodul tagastab vastustes saatmise ajast ühe võrra väiksema väärtuse sama võtmega kodeeritult (s.o. E(CK’, T’-1)). See tõestab, et nõue või vastus pärineb volitatud osapoolelt. Servermoodul lükkab tagasi kõik nõuded, mille korral nende vastuvõtmise aeg on

hilisem kui saatmise aeg pluss akna suurus (T’ + W’). Kliendimoodulil on mõistlik valida akna suurus W nii, et saadetud nõue jõuaks ajavahemiku W jooksul kindlasti servermoodulini, kuid samas oleks W piisavalt väike, et nõude kinnipüüdmiseks ja sellega manipuleerimiseks (korduv saatmine, modifitseerimine vms. tegevus) ei jääks enam piisavalt aega. Lisaks kontrollib servermoodul, et nõude saatmise aeg T’ oleks hilisem kui eelmise nõude saatmise aeg. See välistab volitamata osapoolel nõuete korduva saatmise võimaluse täielikult.

Kliendimoodul omandab kasutaja salajase võtme samas sõlmes töötavalt *võtmeserverilt* (*key server*) [Taylor86]. Võtmeserver omandab salajase võtme aga siis, kui kasutaja end hajusa failisüsteemi vastavasse sõlme logib. Sisselogimisel pöördutakse mingi teenuse (tavaliselt YP või NIS+) serveri poole, mis haldab kasutajate paroolidega kodeeritud salajasi võtmeid, andes ette sisestatud kasutajatunnuse. Selle teenuse server saadab kasutajatunnuse järgi leitud kodeeritud salajase võtme sisselogimisprotsessile. Sisselogimisprotsess dekodeerib kasutaja sisestatud parooliga talle saadetud salajase võtme ning annab selle võtmeserverile üle.

Võtmeserver säilitab omandatud salajase võtme ka pärast seda, kui kasutaja on end hajusa failisüsteemi sõlmest välja loginud, võimaldades kasutajale kuuluvatel klientprotsessidel iseseisvalt töötada. Juhuks, kui klientarvutil peaks toimuma operatsioonisüsteemi iseeneseslik alglaadimine (näiteks ajutise voolukatkestuse tõttu) ja võtmeserver kaotab operatiivmälus hoitud salajased võtmed, on `root`-kasutaja salajane võti salvestatud klientarvuti teatud lokaalsesse faili (enamasti `/etc/.rootkey`), et olulisemad `root`-kasutajale kuuluvad deemonid saaksid normaalselt kaugeid faile kasutada [Taylor86]. Antud lokaalse faili suhtes peaks lugemis- ja kirjutamisõigus olema loomulikult ainult `root`-kasutajal.

2.1.6 Kliendi vahemälu

NFSi kliendimoodul haldab kliendi vahemälu, kus hoitakse kaugete failide ja kataloogide sisu ning atribuute. Viimaseid hoitakse kliendi vahemälu eraldi osas. Kliendi vahemälu kaugeid faile ja katalooge sisaldava osa elementideks on failiblokid.

Uusi elemente paigutatakse vahemällu kõigepealt siis, kui klientprotsess tahab lugeda vahemälust puuduvaid andmeid (süsteemifunktsioonide `read`, `stat` jms. abil),

mistõttu kliendimoodul peab andmete omandamiseks servermooduli poole pöörduma (operatsioonide `read`, `getattr` jms. abil). Failiatribuudid paigutatakse vahemällu ka siis, kui mõni operatsioon (näiteks `read` või `write`) need kõrvalefektina tagastab.

Kui klientprotsess kirjutab süsteemifunktsiooni `write` abil faili, ei saadeta modifitseeritud blokke operatsiooni `write` abil kohe servermoodulile, vaid paigutatakse need samuti vahemällu ning oodatakse, kuni klientprotsess faili süsteemifunktsiooni `close` abil sulgeb. Kui klientarvutil toimub pöördumine süsteemifunktsiooni `sync` poole, saadetakse servermoodulile kõik vahemälus asuvad uuendatud blokid. Katalooge ja failiatribuute modifitseerivate süsteemifunktsioonide (`mkdir`, `link`, `chmod` jms.) väljakutsumisel klientprotsessi poolt pöördub kliendimoodul vastavate operatsioonide abil koheselt servermooduli poole, et erinevates sõlmedes töötavate klientprotsesside ettekujutus kaugete kataloogide sisust ning kaugete failide atribuutidest oleks ühesugune.

Servermooduli olekuvabaduse tagamiseks kasutatakse NFSi korral kliendi juhitud vahemälu valideerimist (muidu peaks servermoodul meeles pidama, millistes kliendi vahemäludes milliseid blokke hoitakse). Selleks jäetakse bloki vahemällu paigutamisel meelde ka aeg, millal blokile vastavat faili viimati modifitseeriti. Bloki kooskõlalisuse kontrollimiseks pöördub kliendimoodul servermooduli poole, loeb operatsiooni `getattr` abil faili atribuudid ja võrdleb meeldejäetud aega atribuutide hulgas oleva faili viimase modifitseerimise ajaga. Kui need erinevad, tunnistatakse blokk kehtetuks.

Kauge kataloogifaili blokkide kooskõlalisust kontrollitakse siis, kui kataloogile viidatakse (näiteks programmiga `ls`). Pärast kontrolli eeldatakse, et kataloogifaili blokid on järgmise 30 sekundi vältel kooskõlalised. Kauge faili blokkide kooskõlalisust kontrollitakse alati siis, kui klientprotsess selle faili süsteemifunktsiooni `open` abil avab, ning siis, kui operatsiooni `read` abil servermoodulilt mõni selle faili blokk loetakse (sel juhul pole operatsiooni `getattr` poole vaja pöörduda, sest operatsioon `read` tagastab ka faili atribuudid, vt. Tabel 2.1). Lisaks võidakse faili blokkide kooskõlalisust kontrollida siis, kui failile viidatakse ning viimasest kontrollist on möödunud üle 3 sekundi. Kliendi vahemälus asuvad failiatribuudid loetakse automaatselt kehtetuks siis, kui nende vahemällu paigutamisest on möödunud 60 sekundit.

Kliendi vahemälude kasutamine põhjustab muidugi kõrvalekaldeid UNIXi failisemantikast failide modifitseerimise osas. UNIXi korral muutuvad kõik mingisse faili tehtud muudatused koheselt nähtavaks kõigile protsessidele, kes on selle faili avanud, NFSi korral on see nii ainult klientprotsessiga samas sõlmes töötavate protsesside jaoks. UNIXi failisemantika paremaks järgimiseks kasutatakse NFSi korral sageli klientarvutites töötavaid *bio-deemoneid* (*bio-daemons*), mis saadavad vahemälus asuva bloki servermoodulile niipea, kui seda on täielikult modifitseeritud [Coul94]. See tähendab, et osa modifitseeritud blokke jõuab servermoodulini juba enne, kui klientprotsess faili süsteemifunktsiooni `close` abil sulgeb.

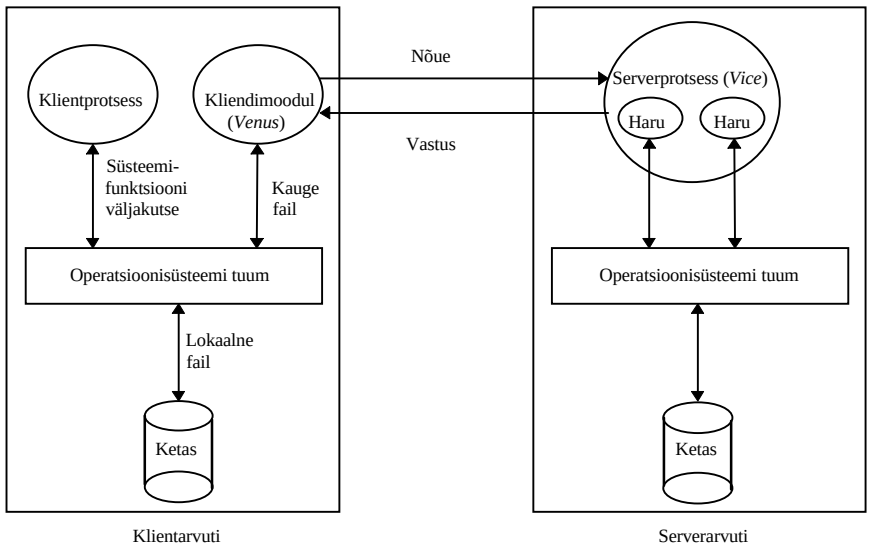
2.2 AFS

AFSi (Andrew File System) väljatöötamist alustati Carnegie Melloni Ülikoolis 1983. aastal. Süsteemi autorite põhieesmärk oli võimalikult skaleeritava hajusa failisüsteemi loomine. 1984. aasta lõpus valmis süsteemi prototüüp AFS-1, 1985. aasta lõpus esimene versioon AFS-2 ning 1989. aastal kommertsproduktina AFS-3, mille loomine toimus tarkvarafirma Transarc Corporation toetusel. 1991. aastal koosnes Carnegie Melloni Ülikoolile kuuluv AFSi installatsioon umbes 800 klientarvutist ning 40 neid teenindavast serverarvutist, samuti eksisteerib AFS-3 ülemaailmse ulatusega installatsioon, mille tuhanded sõlmed asuvad erinevatel kontinentidel. See näitab, et autoritel õnnestus saavutada oma eesmärk - luua võimalikult skaleeritav hajus failisüsteem. Selles punktis käsitletakse põhiliselt AFS-2 ülesehitust.

2.2.1 Kliendimoodul ja serverprotsess

AFSi tarkvara kirjutati (sarnaselt NFSiga) algul UNIX-operatsioonisüsteemiga arvutite jaoks ning ka praegu kasutatakse seda enamasti koos UNIXiga. Maksimaalse ligipääsu läbipaistvuse saavutamiseks on klientarvuti operatsioonisüsteemi tuuma modifitseeritud, kuid kliendimoodul (*Venus*) on realiseeritud siiski eraldi protsessina (AFS-3 puhul paigutati ta töökiiruse tõstmiseks operatsioonisüsteemi tuuma). Kui kaugele failile rakendatakse mingit süsteemifunktsiooni, edastab tuum vajalikud parameetrid kliendimoodulile, mis omakorda pöördub kaugprotseduuri väljakutsega

serverprotsessi (*Vice*) poole, saates nõude serverarvuti kokkulepitud porti. Serverprotsess võtab kindlaksmääratud pordist vastu kliendimoodulite pöördumisi ning loob iga nõude töötluseks omaette *protsessiharu* (*thread*) (vt. Joonis 2.2). AFS-1 korral loodi selleks omaette protsess, kuid kuna protsessidevahelisel lülitumisel toimuvad kontekstivahetused koormasid serverarvuti protsessorit liialt, sellest lähenemisest loobuti [Howard88].



Joonis 2.2 AFS.

2.2.2 Kliendi vahemälu

AFSi kliendimoodul haldab kliendi vahemälu, kus hoitakse kaugete failide, kataloogide ja sümbolviidete sisu ning atribuute. AFSi põhiline omapära võrreldes teiste hajusate failisüsteemidega seisneb selles, et kliendi vahemälu elementideks on terved failid. Kuna AFSi loomise ajal sedalaadi vahemälu tüüpilise tööjaama operatiivmällu ei mahtunud, otsustasid AFSi autorid vahemälu klientarvuti kettale paigutada. Atribuute sisaldav kliendi vahemälu osa asub siiski operatiivmälus, et süsteemifunktsiooni `stat` täitmisele kuluks vähem aega [Howard88].

AFS-1 korral hoitakse kliendi vahemälus vaid kaugeid faile ja nende atribuute. Kui klientprotsess avab faili, kontrollib kliendimoodul, kas selle faili koopia on juba vahemälus. Kui see vahemälus leidub, pöördub kliendimoodul koopia kooskõlalisuse

kontrolliks serverprotsessi poole, võrreldes koopia atribuute serverarvutil asuva faili omadega. Kui koopia on mittekooskõlaline või seda vahemälus pole, loeb kliendimoodul faili ja selle atribuudid vahemällu. Et vähendada serverarvuti ja sellega ühendatud kohtvõrgu koormust, viiakse edasised selle failiga tehtavad operatsioonid läbi vahemälus asuval koopial ning mingeid uusi pöördumisi serverprotsessi poole koopia kooskõlalisuse kontrolliks ja uute blokkide lugemiseks ei toimu. Kui klientprotsess faili modifitseerib, siis faili sulgemisel klientprotsessi poolt saadab kliendimoodul koopia koos atribuutidega serverprotsessile.

AFSi autoritele tundus, et AFS-1 pole siiski piisavalt skaleeritav. Et vähendada pöördumisi serverprotsessi poole ning suurendada sellega süsteemi skaleeritavust koormuse suhtes, hoitakse AFS-2 korral kliendi vahemälus ka kaugete kataloogide ja sümbolviidete sisu ning atribuute [Howard88]. Kui klientprotsess avab kataloogifaili lugemiseks või viitab sümbolviitele ning kataloogifaili või sümbolviite koopiat vahemälus ei ole, pöördub kliendimoodul serverprotsessi poole ning loeb kataloogifaili või sümbolviite sisu vahemällu. Kataloogi või failiatribuutide modifitseerimisel (süsteemifunktsioonide `mkdir`, `link`, `chmod` jms. abil) pöördub kliendimoodul vastava operatsiooni abil koheselt serverprotsessi poole, et kõigi klientprotsesside ettekujutus kaugete kataloogide sisust ja kaugete failide atribuutidest oleks ühesugune.

Et suurendada süsteemi skaleeritavust veelgi, kasutatakse AFS-2 korral serveri juhitud vahemälu valideerimist [Howard88]. Serverprotsess peab meeles, millised kliendimoodulid milliseid faile ja failiatribuute vahemälus hoiavad. Kui kliendimoodul loeb faili koos atribuutidega (või ainult faili atribuudid) serverarvutilt oma vahemällu, annab serverprotsess talle selle faili kohta *garantii* (*callback*). Kui faili või selle atribuute modifitseeritakse ning tehtud muudatused serverprotsessini jõuavad, saadab serverprotsess ülejäänud seda faili ja selle atribuute (või ainult selle atribuute) vahemälus hoidvatele kliendimoodulitele teate, et nende valduses olevad faili ja atribuutide koopiad on mittekooskõlalised (ehk *tühistab garantii*). Kui klientprotsess avab faili või viitab failiatribuutidele, kontrollib kliendimoodul nüüd ainult, kas vahemälus leidub serverprotsessi poolt garanteeritud koopia. Kui avatavat või viidatavat objekti vahemälus pole või selle garantii on tühistatud, pöördub kliendimoodul uue koopia omandamiseks serverprotsessi poole. Kui faili kohta käiv garantii tühistatakse pärast faili avamist selle kasutamise ajal, kliendimoodul faili uuesti

vahemällu ei loe; kui faili kasutamise ajal modifitseeritakse, siis selle sulgemisel saadab kliendimoodul faili koopia serverprotsessile nagu harilikult.

Selline lähenemine vähendab serverarvuti ja sellega ühendatud kohtvõrgu koormust veelgi, sest kaugete failide kasutamisel pöördutakse serverprotsessi poole ainult siis, kui selleks on vajadus. Samas tekivad mõned probleemid. Kui serverprotsessi teade garantii tühistamisest ei jõua mingil põhjusel kliendimoodulini, võivad klientprotsessid pikemat aega mittekooskõlaliste andmetega töötada. Selle vältimiseks on AFS-2 hiljem täiendatud, loobudes *puhtast* serveri juhitud vahemälu valideerimisest. Kui faili avamisel või failiatribuutidele viitamisel on garantii andmisest möödunud ajavahemik T ning serverprotsessiga pole selle aja jooksul kontakti olnud, siis pöördub kliendimoodul vahemälus asuva koopia kooskõlalisuse kontrolliks ise serverprotsessi poole. T on tüüpiliselt pikem ajavahemik (tavaliselt 10 minutit) [Coul94].

AFS-2 korral ei ole serverprotsess enam olekuvaba - ta peab meeles, millised kliendimoodulid milliseid faile ja failiatribuute vahemälus hoiavad. Avariilõpujärgsete info taastamisprotseduuride vältimiseks serverprotsessi taaskäivitamisel hoitakse sedalaadi infot operatiivmälu asemel serverarvuti kettal [Coul94]. Kui vaba kettamälu hulk on vähenenud teatud piirini, tühistab serverprotsess osa kliendimoodulitele antud garantiidest ja vabastab vastava mälu [Howard88]. See suurendab süsteemi skaleeritavust populatsiooni suhtes. Olekuvabaduse puudumine on mõnes mõttes positiivne, sest nüüd on võimalik serverprotsessi operatsioonidena realiseerida näiteks kaugete failide lukustamine. Olekuvaba serverprotsessi korral seda teha ei saa, sest serverprotsess peab meeles pidama seatud lukke ja nende omanikke, mis läheb olekuvabaduse põhimõttega vastuollu.

2.2.3 Volüümid

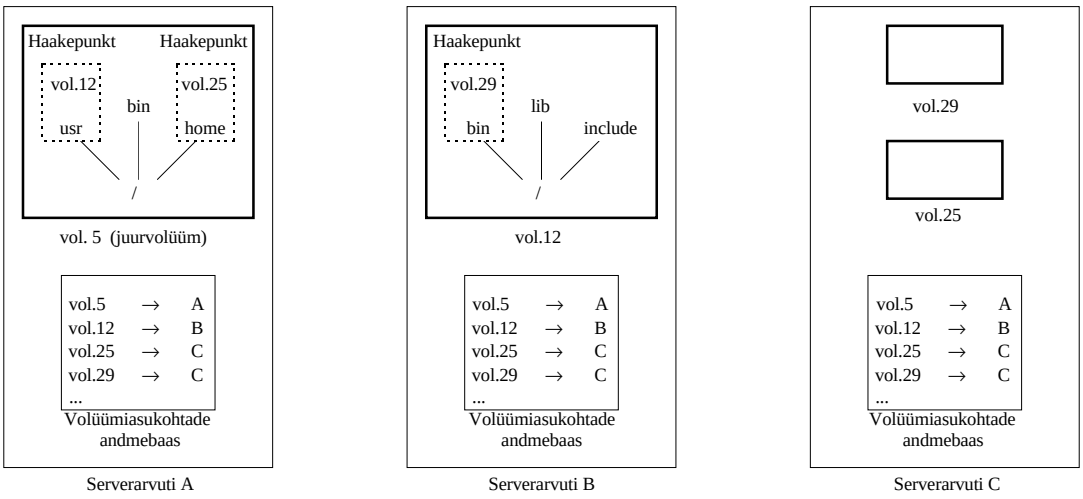
AFS-2 puhul on kauged failid hajusa failisüsteemi administreerimise hõlbustamiseks *volüümidesse* (*volumes*) koondatud [Howard88]. Volüüm on vähim loogiline ühik, mida serverprotsess võib hallata. Tavaliselt sisaldab serverarvuti ketta iga partitsioon paljusid volüüme, kuid samas pole volüüm seotud ühegi partitsiooniga, vaid tema asukoht võib muutuda. AFS-1 korral on haldusühikuteks serverarvuti ketta

partitsioonid. Kuna volüümid on partitsioonidest tunduvalt väiksemad ning ketta füüsilisest struktuurist sõltumatud, muudab nende kasutamine kaugete failide ümberpaigutamise ühelt serverarvutilt teisele lihtsamaks. AFS-1 korral tuleb ühe haldusühiku ümberpaigutamiseks teisele serverarvutile luua selle kettal eraldi partitsioon, volüümi võib aga paigutada suvalisse juba eksisteerivasse partitsiooni, kus on piisavalt ruumi.

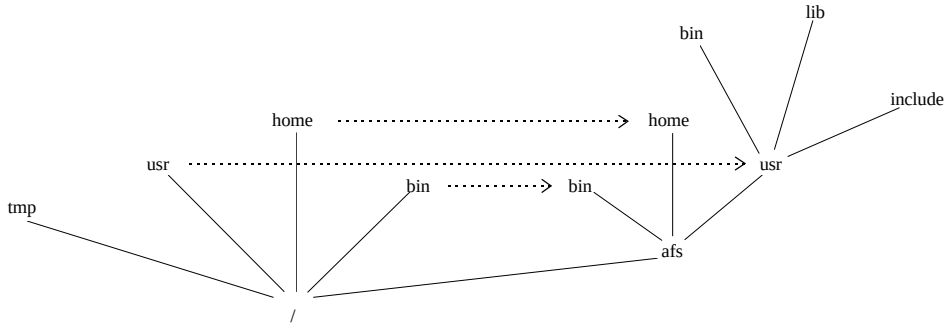
Kõik kauged failid on erinevatel serverarvutitel paiknevate volüümide vahel ära jagatud, millest üks on *juurvolüüm* (*root volume*). Serverarvutite ketastel asuvate failide ja kataloogide hierarhiliseks organiseerimiseks haagitakse juurvolüümi mingitesse kataloogidesse mõned teised volüümid, kusjuures nende volüümide kataloogid võivad omakorda olla haakepunktideks kolmandatele volüümidele jne. Haakepunkt kujutab endast pseudokataloogi - ta sisaldab ainult haagitud volüümi identifikaatorit, tegelik sisu tal puudub (vt. Joonis 2.3 ja Joonis 2.4). Haakimine seisneb sisuliselt haakepunkti loomises süsteemiadministraatori poolt. Erinevalt NFSist on siin kasutatav meetod *tsentraliseeritud*, sest kogu haakimist puudutav info asub serverarvutitel - mingisse volüümi haakepunkti loomisega muutub sellega läbiviidud haakimine koheselt nähtavaks kõigis klientarvutites töötavatele klientprotsessidele. Tsentraliseeritud meetodi kasutamine suurendab hajusa failisüsteemi skaleeritavust administreerimise suhtes, sest haakimisel tehtav administratiivne töö ei sõltu süsteemi suurusest nagu mittetsentraliseeritud meetodi korral [Sat93]. Juurvolüüm on kõigil klientarvutitel seotud kokkuleppeliselt kataloogiga `/afs`. Seega garanteerib AFS[4] täieliku asukoha läbipaistvuse, sest failinimede ruum on kõigi klientprotsesside jaoks ühesugune [Sat93].

Igal AFSi serverarvutil asub andmebaas, kus volüümide identifikaatoritele seatakse vastavusse serverarvutite nimed, mille ketastel need volüümid asuvad (vt. Joonis 2.3). Ühele volüümi identifikaatorile võib vastata mitu serverarvuti nime, mis võimaldab vajadusel replikatsiooni kasutada. Kuna AFSis puuduvad mehhanismid muudatuste kiireks levitamiseks volüümi erinevatel serverarvutitel asuvate koopiate vahel, on replikatsioon oma olemuselt primitiivne - teda saab segadust tekitamata kasutada vaid harva modifitseeritavate volüümide korral.

[4] Edaspidi tähistab nimi AFS kogu töös AFS-2.



Joonis 2.3 Volüümid ja volüümiasukohtade andmebaasid (haakepunktid on ümbritsetud katkendliku joonega, volüümid jämeda joonega).



Joonis 2.4 Suvalisele klientprotsessile nähtav kataloogipuu (katkendlike nooltega on tähistatud sümbolviited, mida võidakse igal klientarvutil failide mittelokaalsuse varjamiseks luua).

Volüüme on võimalik erilise operatsiooni abil ühelt serverarvutilt teisele paigutada, ilma et klientprotsessid seda märkaksid [Howard88]. Enne volüümi ümberpaigutamist luuakse *kloon* (*clone*), mis kajastab täielikult volüümi sisu antud hetkel, ning saadetakse see uuele serverarvutile, kus klooni põhjal volüüm taasluuakse. Kui saatmise käigus peaks mõni klientprotsess volüümi tema vanas asukohas uuendama, saadetakse muudatused uude asukohta *inkrementaalse klooni* (*incremental clone*) näol. Kui ka selle saatmise ajal tehakse volüümi muudatusi, saadetakse ka need uude asukohta jne., kuni mingi saatmise käigus volüümi sisus enam muutusi ei toimu. Pärast seda modifitseeritakse vastavalt volüümiasukohtade andmebaasi. Kuna andmebaasi tehtud muudatus ei pruugi kohe kõigi serverarvutiteni jõuda, võib volüümi enne hallanud

serverprotsess saabunud nõuded uuele haldajale ümber suunata. Volüümi ümberpaigutamise operatsioon on klientprotsessidele nähtamatu ja *atomaarne* (*atomic*) - ta kas viiakse täielikult läbi või ei ole tal mingisugust efekti. Ükskõik kumma ümberpaigutamises osaleva serverprotsessi töö avariilõpu korral operatsiooni läbiviimine katkestatakse ja volüüm jääb endiselt vanast asukohast kättesaadavaks.

2.2.4 Serverprotsessi liides

Olulisemad AFSi serverprotsessi operatsioonid on toodud Tabel 2.3-s [Howard88].

`Fetch`	Tagastatakse faili (või kataloogi) atribuudid või atribuudid ning sisu. Serverprotsessilt võetakse mõlemal juhul faili (või kataloogi) kohta garantii.
`Store`	Faili atribuudid või atribuudid ning sisu saadetakse serverprotsessile.
`Remove`	Kustutatakse fail.
`Create`	Luuakse uus fail, serverprotsessilt võetakse selle faili kohta garantii.
`Rename`	Muudetakse faili või kataloogi nime (sama volüümi piires).
`Symlink`	Luuakse sümbolviide failile või kataloogile.
`Link`	Luuakse viide failile (viide peab asuma samas kataloogis).
`Makedir`	Luuakse kataloog.
`Removedir`	Kustutatakse tühi kataloog.
`SetLock`	Failil või kataloogil seatakse jagatav või välistav lukk. Lukk aegub 30 minuti möödudes.
`Releaselock`	Kõrvaldatakse seatud lukk.
`GetRootVolume`	Tagastatakse juurvolüümi identifikaator.
`GetVolumeInfo`	Tagastatakse serverarvutite nimed, mille ketastel antud volüüm asub.
`RemoveCallBack`	Teatatakse failile või kataloogile antud garantii mittevajalikkusest. Kliendimoodul pöördub selle operatsiooni poole pärast faili või kataloogi vahemälust välja jätmist.
`BreakCallback`	Serverprotsess pöördub kliendimooduli poole, teatades mingile failile või kataloogile antud garantii tühistamisest.

Tabel 2.3 AFSi serverprotsessi olulisemad operatsioonid.

Failinimede interpreteerimine toimub kliendimoodulis operatsiooni `Fetch` abil, kus nimi osadeks jagatakse ja iteratiivselt lahendatakse. Kui mingile nimeosale vastav kataloogifail on vahemälus ning garanteeritud, loeb kliendimoodul järgmise nimeosa

identifikaatori sellest kataloogifailist, vastasel korral loeb ta kõigepealt nimeosale vastava kataloogifaili operatsiooni `Fetch` abil vahemällu. Kui nime lahendamisel jõutakse haakepunktini, siis on kliendimoodulil võimalik haakepunktis sisalduva volüümi identifikaatori järgi leida serverarvuti nimi, mille kettal volüüm asub, kasutades operatsiooni `GetVolumeInfo` abil suvalisel serverarvutil leiduvat volüümiasukohtade andmebaasi.

2.2.5 Kasutajad, failide kaitse ja UNIXi failisüsteemi emuleerimine

AFSi kasutajad on koondatud *gruppidesse*, kusjuures grupid võivad omakorda sisaldada teisi gruppe [Sat89]. Kõigi kasutajate ja gruppide hulka nimetatakse *kaitsedoomeniks* (*protection domain*). Igal kasutajal ja grupil on tekstiline nimi, süsteemisisemiselt vastab nimele 32-bitine identifikaator. Iga grupp kuulub mingile kasutajale, kelle tekstiline nimi koos kooloniga peab asuma grupinime alguses.

Põhimõte, et grupid võivad sisaldada teisi gruppe, lihtsustab paljude kasutajatega suurema hajusa failisüsteemi administreerimist. Kasutaja ühte gruppi lisamisega (või sealt eemaldamisega) võib vajadusel saavutada antud kasutaja lisamise mitmesse gruppi (või eemaldamise mitmest grupist). Gruppi on võimalik ühe operatsiooniga lisada (või sealt eemaldada) mingit ühist tunnust omavaid ning ühte gruppi kuuluvaid kasutajaid. Kuna hajusa failisüsteemi laienemisel kasvab tõenäoliselt ka kasutajate ja gruppide arv, muudab selline lahendus süsteemi administreerimise suhtes skaleeritavamaks.

On olemas eriline pseudokasutaja `System` ning eriline administraatorigrupp `System:Administrators`. AFSi administreerimisega saavad tegeleda ainult sellesse gruppi kuuluvad tõelised kasutajad, pseudokasutajal `System` on see aga keelatud (erinevalt UNIXist, kus süsteemi administreerimine toimub pseudokasutaja `root` nime all). Selline lähenemine võimaldab auditiprogrammidel teha kindlaks konkreetne isik, kes on süsteemi administreerinud. Administreerimisõiguse äravõtmiseks tuleb kasutaja grupist `System:Administrators` kõrvaldada, mis on tunduvalt lihtsam kui pseudokasutaja `System` parooli vahetamine ja selle teadaandmine kõigile ülejäänud administraatoritele. Hajusa failisüsteemi julgeoleku seisukohalt on selline lähenemine

tunduvalt turvalisem, sest süsteemi suurenedes administraatorite arv tõenäoliselt kasvab ning suureneks tõenäosus, et pseudokasutaja `System` parool tuleb avalikuks.

Iga AFSi kasutaja kuulub automaatselt gruppi `System:AnyUser`, mis ei tohi omakorda ühessegi teise gruppi kuuluda. Mingi kasutaja X (või grupi X) *kehtivaks kaitsedoomeniks* (*current protection subdomain*) nimetatakse hulka, kuhu kuulub X ise ning kõik grupid, kuhu X otseselt või kaudselt (s.t. teiste gruppide kaudu) kuulub.

Kaitsedoomenit esitav andmebaas asub igal AFSi serverarvutil. Selle andmebaasi iga kirje sisaldab mingi kasutaja (või grupi) X nime ja 32-bitist identifikaatorit, kõiki gruppe, kuhu X otseselt kuulub ning kasutaja (või grupi) X kehtivat kaitsedoomenit. Kui X on kasutaja, on kirjes loetletud veel grupid, mille omanik X on; kui X on grupp, siis kasutajad, kes kuuluvad gruppi X otseselt. Kaitsedoomenit esitava andmebaasi uuendamiseks modifitseeritakse kokkulepitud serverarvuti juures asuvat *põhikoopiat* (*master copy*), kust levitatakse muudatusi teistel serverarvutitel paiknevatele koopiatele.

Kaugete failide kaitsmiseks volitamata ligipääsu eest on iga kauge kataloogiga seotud juurdepääsunimekiri. Nimekiri laieneb kõigile kataloogi failidele - AFSi korral on juurdepääsunimekirjad ainult kataloogidel [Sat89]. Seetõttu väheneb oluliselt failiatribuutides sisalduv info, mida hoitakse vahemäludes ja serverarvutite ketastel ning mis liigub kommunikatsioonikanalite kaudu kliendimoodulite ja serverprotsesside vahel. Kui kataloogi mingil failil on vaja eraldi juurdepääsunimekirja, tuleb see fail omaette kataloogi paigutada ning luua esialgsesse kataloogi antud failile osutav sümbolviide.

Juurdepääsunimekirja iga element koosneb kasutaja või grupi X nimest ning tema *õiguste loetelust* (*rights list*) antud kataloogi suhtes (vt. Tabel 2.4).

`r`	õigus lugeda suvalist kataloogis paiknevat faili.
`w`	õigus kirjutada suvalisse kataloogis paiknevasse faili.
`l`	õigus lugeda suvalise kataloogis paikneva faili atribuute.
`i`	õigus luua kataloogi uusi faile.
`d`	õigus kustutada suvalist kataloogis paiknevat faili.
`a`	õigus muuta kataloogi juurdepääsunimekirja.
`k`	õigus lukustada suvalist kataloogis paiknevat faili.

Tabel 2.4 Kasutaja või grupi võimalikud õigused kataloogi suhtes.

Õiguste loetelus on ära toodud *positiivsed* ja *negatiivsed õigused* (*positive and negative rights*) [Sat89]. Positiivne õigus tähendab õiguse omamist, negatiivne aga õiguse puudumist. Konflikti korral on negatiivsel õigusel prioriteet. Kasutaja või grupi õiguste leidmiseks mingi kataloogi suhtes leiab serverprotsess kõigepealt kasutaja või grupi kehtiva kaitsedoomeni elementide positiivsete õiguste summa. Seejärel leiab ta vastava negatiivsete õiguste summa ning lõpuks lahutab positiivsete õiguste summast negatiivsete õiguste summa. Kasutaja või grupi kehtiva kaitsedoomeni saab serverprotsess leida serverarvutil asuvast kaitsedoomenit esitavast andmebaasist.

Negatiivsete õiguste kasutamine suurendab jällegi süsteemi skaleeritavust administreerimise suhtes. Kui näiteks mingi kasutaja otsustatakse grupist kõrvaldada, võib muudatuste levimine kaitsedoomeni andmebaasi põhikoopialt kõigi ülejäänud koopiateni suurema hajusa failisüsteemi puhul (kuhu kuulub palju serverarvuteid) palju aega võtta. Enne muudatuste lõplikku jõustumist võib oluliste kataloogide korral, millele grupil on teatud õigus, seada grupist kõrvaldatava kasutaja jaoks vastava negatiivse õiguse.

UNIXi failisüsteemi emuleerimine on nagu NFSi korralgi osaliselt kliendimooduli ülesanne. Kuna UNIXi ja AFSi mehhanismid failide kaitsmiseks on erinevad, ei õnnestu emulatsioon selles osas siiski täielikult. UNIXi 9-bitine faili *kaitsekood* ei oma AFSi korral enam endist tähendust. Failide korral on olulised vaid failiomaniku õigusi näitavad 3 bitti, kataloogide puhul ei ole ka neil tähtsust [Sat89]. Klientarvutitesse installeeritud programmi `ls` on modifitseeritud nii, et kauge kataloogi kaitsekoodi ei näidata üldse ning kaugete failide korral näidatakse ainult failiomaniku õigusi kajastavaid bitte. Need 3 bitti on küll faili atribuutide hulgas ja seega serverarvuti kettale salvestatud, kuid neil on tähendus vaid kliendimooduli jaoks, kui fail on koos atribuutidega kliendi vahemällu loetud. Näiteks omanik võib kauge faili kaitsekoodiks seada `r--`, et hoiduda faili eksikombel kustutamast. Samas ei oma kaitsekood klientprotsessi õiguste kontrollimisel serverprotsessi jaoks mingit tähendust.

UNIXi failisemantika täpse järgimise seisukohalt põhjustab probleeme see, kui kataloogi failile laienev juurdepääsunimekiri pärast faili avamist muutub. Kui klientprotsess kaotab kirjutamisõiguse ja sulgeb pärast seda faili, ei arvesta

serverprotsess faili tehtud muudatusi ning operatsioon `Store` tagastab veakoodi [Sat89]. UNIXi korral lubataks fail siiski sulgeda ja faili tehtud muudatused läheksid arvesse; veasituatsioon tekiks alles järgmisel avamisel. Selle probleemi lahendamine kliendimooduli tasemel on võimatu. Klientprogrammid tuleb kirjutada nii, et alati kontrollitaks süsteemifunktsiooni `close` poolt tagastatud väärtusi ning veasituatsiooni tekkimisel astuks vajalikke samme klientprotsess ise.

2.2.6 Autentimine

On oluline, et AFSi serverprotsess ja kasutaja oleksid üksteise autentsuses kindlad ning et nendevaheline kommunikatsioon oleks turvaline - oleks välistatud teadete pealtkuulamine ja modifitseerimine. Selleks kasutatakse autentimisprotokolli, mille tulemusena kasutaja ja serverprotsess veenduvad üksteise autentsuses ning omandavad salajase võtme SK, mille abil üksteisele saadetavaid teateid kodeerida. Kuna AFSi installatsioon võib (erinevalt NFSist) koosneda tuhandetest sõlmedest ja paljude organisatsioonide kohtvõrkudest, avarduvad sellega ka võimalused võrgu pealtkuulamiseks. Seetõttu kodeeritakse kaugprotseduuride väljakutsetel (erinevalt NFSist) nii nõue kui vastus täielikult [Sat89, Sat93], mistõttu kliendimooduli ja serverprotsessi vahelise kommunikatsiooni pealtkuulamine pole enam võimalik.

Autentimisel esindab kasutajaid kliendimoodul. Autentimisprotokolli (vt. Tabel 2.5) parameetriteks on *protokollivõti* (*handshake key*) `HKC`, kliendiidentifikaator `ClientID` ning protseduur `GetKeys` [Sat89].

AuthenticationProtocol (HKC, ClientID, GetKeys)
1. Kliendimoodul genereerib juhusliku täisarvu X ja kodeerib selle protokollivõtmega HKC. Tulemus E(HKC, X) koos kodeerimata kliendiidentifikaatoriga ClientID saadetakse serverprotsessile.
2. Teate [E(HKC, X), ClientID] saamisel pöördub serverprotsess protseduuri GetKeys poole, andes parameetrina ette teatest omandatud kliendiidentifikaatori ClientID. GetKeys tagastab kaks võtit, millest üks peab olema võtmega HKC identne võti HKS ja teine *sessioonivõti* (*session key*) SK, mida hakatakse kasutama autentimisjärgse kommunikatsiooni vältel. On garanteeritud, et kellelgi peale serverprotsessi ei õnnestu võtit HKS omandada. Kui protseduuri GetKeys täitmine ebaõnnestub (näiteks seetõttu, et parameetrile ClientID ei vasta ühtegi võtit), lükatakse kliendimooduli pöördumine tagasi ja autentimine lõpeb.
3. Serverprotsess leiab saadud teatest võtme HKS abil arvu X' = D(HKS, E(HKC, X)). Kui HKS = HKC, siis ka X' = X.
4. Serverprotsess arvutab X' + 1 ning genereerib juhusliku täisarvu Y, kodeerib mõlemad võtmega HKS ning saadab tulemuse E(HKS, [X' + 1, Y]) kliendimoodulile.
5. Kliendimoodul dekodeerib saadud teate võtme HKC abil ning leiab talle saadetud arvupaari [Z, W] = D(HKC, E(HKS, [X' + 1, Y])). Kui Z = X + 1, siis loetakse serverprotsessi autentsus tõestatuks, sest ainult serverprotsess võis leida võtme HKS ning selle abil talle saadetud arvu X, viimast õigesti teisendada ning tulemuse korrektselt kodeerida. Arvu X protokollis ettenähtud teisendus on vajalik, et volitamata osapool ei saaks kodeeritud arvu X kopeerida serverprotsessile sammu 1 kohaselt saadetud teatest.
6. Kliendimoodul arvutab W + 1, kodeerib selle võtmega HKC ning saadab tulemuse E(HKC, W + 1) serverprotsessile.
7. Serverprotsess leiab talle saadetud arvu W' = D(HKS, E(HKC, W + 1)). Kui W' = Y + 1, siis loetakse kasutaja autentsus tõestatuks, sest ainult teda esindav kliendimoodul võis teada võtit HKC ning leida selle abil talle saadetud arvu Y, viimast õigesti teisendada ning tulemuse korrektselt kodeerida. Arvu Y protokollis ettenähtud teisendus on vajalik, et volitamata osapool ei saaks kodeeritud arvu Y kopeerida kliendimoodulile sammu 4 kohaselt saadetud teatest.
8. Serverprotsess kodeerib sessioonivõtme SK ja genereeritud juhusliku täisarvu N võtmega HKS ja saadab tulemuse E(HKS, [SK, N]) kliendimoodulile. Kogu edasise kasutajale kuuluvate klientprotsesside ja serverprotsessi vahelise kommunikatsiooni vältel kasutatakse kaugprotseduuride väljakutsetel saadetavate nõuete ja vastuste kodeerimiseks võtit SK. Nõuded ja vastused kannavad monotoonselt kasvavaid järjekorranumbreid alates arvust N. Viimane abinõu tagab, et kui volitamata osapool teateid korduvalt saadab, on kliendimoodulil ja serverprotsessil võimalik seda kindlaks teha ning selliseid teateid ignoreerida.

Tabel 2.5 Autentimisprotokoll.

Ülalkirjeldatud autentimisprotokolli kasutatakse kõigepealt siis, kui kasutaja logib end klientarvutisse. Autentimine toimub *autentimisserveri* (*authentication server*) ja kasutaja vahel, kliendiidentifikaatorina kasutatakse sisestatud kasutajatunnust ning protokollivõtmena kasutaja parooli. Autentimisserver haldab paroolide andmebaasi, kus paroolid on (erinevalt UNIXist ja NFSist) kodeerimata. See tähendab, et peab olema tagatud selle sõlme füüsiline julgeolek, kus autentimisserver töötab. Protseduur `GetKeys` seisneb kasutajatunnuse järgi parooli leidmises ning sessioonivõtme genereerimises.

Kohe pärast autentimisserveri ja kasutaja vahelise autentimise lõppu saadab autentimisserver kliendimoodulile kui kasutaja esindajale *lubatähe* (*token*) nii kodeeritud kui kodeerimata kujul (kodeeritud ja kodeerimata lubatähte sisaldav teade kodeeritakse enne saatmist autentimise käigus genereeritud sessioonivõtmega). Lubatäht sisaldab kasutaja 32-bitist identifikaatorit, võtit K, mida kasutatakse hiljem failiserveritega kontakteerumisel, ning kellaaegu, mille vahel lubatäht kehtib. Kodeeritud lubatähe on autentimisserver kodeerinud võtmega, mis on teada ainult autentimisserverile ja failiserveritele. Kodeeritud ja kodeerimata lubatäht jäävad kliendimooduli valdusse, kes kasutab neid antud kasutaja esindamisel.

Lisaks sellele kasutatakse autentimisprotokolli kasutaja ja failiserveri vahel, kui kasutajale kuuluval klientprotsessil on vaja opereerida serverprotsessi halduses olevate failidega, kuid antud kasutaja ja serverprotsessi vahel pole autentimist veel toimunud. Kliendiidentifikaatorina kasutab kliendimoodul autentimisserverilt saadud kodeeritud lubatähte ning protokollivõtmena autentimisserverilt saadud võtit K. Protseduur `GetKeys` seisneb järgmistes tegevustes: serverprotsess dekodeerib kodeeritud lubatähe võtmega, mis on teada ainult autentimis- ja failiserveritele, omandab lubatähest võtme K ja kasutaja 32-bitise identifikaatori ning genereerib sessioonivõtme.

Kui lubatähes näidatud ajavahemik möödub ning lubatäht aegub, tuleb uuesti autentimisserveri poole pöörduda. Lubatähes näidatud ajavahemik on tüüpiliselt 24 tunni pikkune ning seetõttu piisab enamasti ühekordsest pöördumisest sisselogimisel.

3. Vahemälud

Vahemälude kasutamine on üks olulisemaid võtteid, mille abil hajusa failisüsteemi skaleeritavust[5] ja klientprotsesside töökiirust suurendatakse. Nagu eelpool mainitud (vt. lk. 3), ei leidu tänapäeval ühtegi tõsiseltvõetavat hajusat failisüsteemi, kus seda võtet ei rakendataks. Muidugi on vahemälude kasutamisel oma hind - failioperatsioonide semantika ei ole mõnikord traditsioonilise failisemantikaga identne. Näiteks leidub hajusaid failisüsteeme, kus klientprotsessi poolt faili tehtud muudatusi hoitakse kuni faili sulgemiseni ainult kliendi vahemälus, mistõttu pole need mõne aja jooksul teistele klientprotsessidele nähtavad. Samas peaks näiteks UNIXi failisemantika kohaselt saama suvaline faili kirjutamine igale klientprotsessile nähtavaks vahetult pärast kirjutamisoperatsiooni lõppu. Praktika on siiski näidanud, et mõningane eemaldumine traditsioonilisest failisemantikast ei häiri hajusa failisüsteemi kasutajaid mainimisväärselt.

3.1 Sissejuhatus

3.1.1 Failikasutusmallid

Hajusa failisüsteemi vahemälude disainimisel tuleb arvestada failide kasutamise iseärasusi ehk *failikasutusmalle* (*file access patterns*). Kui näiteks osutub, et pärast faili avamist opereerivad klientprotsessid enamasti kogu faili sisuga, on kliendi vahemälu elementidena mõtet kasutada terveid faile, et vältida korduvaid pöördumisi serverprotsessi[6] poole üksikute blokkide lugemiseks. Failikasutusmallide avastamiseks alustati esimesi uuringuid 1980-ndate aastate esimesel poolel. Algul uuriti, millised on failikasutusmallid mittehajusate operatsioonisüsteemide korral, sest sel ajal veel puudusid laiemalt kasutatavad hajusad failisüsteemid, millest kogutud andmete põhjal oleks saanud tõsiseltvõetavaid järeldusi teha. Uuringute autorid oletasid, et hajusate

[5] Termini *skaleeritavus* all mõeldakse selles peatükis *skaleeritavust koormuse suhtes*, kuna skaleeritavuse ülejäänud aspekte (vt. lk. 13) puudutatakse selles peatükis väga põgusalt.
[6] Edaspidi kasutatakse lihtsuse mõttes kõikjal terminit *serverprotsess*, kuigi mõnes hajusas failisüsteemis pakub failiteenust serverarvuti operatsioonisüsteemi tuumas realiseeritud *servermoodul*.

failisüsteemide korral jäävad failikasutusmallid samaks. Tuntuim töö sellest valdkonnast on John Ousterhouti 1985. aastal ilmunud “A Trace-Driven Analysis of the UNIX 4.2 BSD File System” [Oust85]. Selles töös esitatud väiteid failide kasutamise iseloomu kohta on arvestatud paljude hajusate failisüsteemide (AFS, Sprite jt.) vahemälude disainimisel [Nelson87, Howard88]. 1991. aastal uuriti, millised on failikasutusmallid hajusa failisüsteemi Sprite korral, ning saadud tulemused kinnitasid John Ousterhouti 1985. aastal esitatud väiteid [Baker91].

Tähtsamad paljude uuringute käigus avastatud failikasutusmallid on järgmised:

1. *Ajaline lokaalsus* (*temporal locality*). Kui viidatakse failis sisalduvatele andmetele, siis suure tõenäosusega viidatakse neile lühikese aja pärast uuesti [Sat93]. Tüüpiliselt kasutatakse faili lühema ajavahemiku jooksul palju kordi, millele järgneb pikem ajavahemik, kus faili üldse ei kasutata.
2. *Lugemiste domineerimine*. 2/3 kõigist failide avamistest on avamised lugemiseks [Oust85, Thom87]. Mõnede uuringute käigus on koguni leitud, et üle 80% avamistest on avamised lugemiseks [Baker91, Blaze93].
3. *Failide kogu sisu kasutamine*. Pärast faili avamist opereeritakse tavaliselt kogu faili sisuga. John Ousterhouti mõõtmiste põhjal on 2/3 lugemiseks avamistest sellised, mille järel loetakse faili kõiki baite, ning 80% kirjutamiseks avamistest sellised, mille järel fail täielikult üle kirjutatakse [Oust85].
4. *Failide järjestikune kasutamine*. Pärast avamist kasutatakse faili enamasti *järjestikuselt* (*sequentially*) - lugemine või kirjutamine algab mingist failipositsioonist ning liigutakse faili lõpu suunas ilma ühtegi baiti vahele jätmata. On harv juhus, kui üksikute lugemis- või kirjutamisoperatsioonide vahel failiakent edasi või tagasi nihutatakse. Üle 90% lugemiseks avamistest ja üle 95% kirjutamiseks avamistest on sellised, millele järgneb faili järjestikune kasutamine [Oust85, Baker91]. Samas, kui fail avatakse *nii* lugemiseks *kui ka* kirjutamiseks, siis rohkem kui 75% juhtudest ei kasutata faili järjestikuselt [Oust85, Baker91]. Seda tüüpi avamised esinevad aga väga harva [Oust85, Baker91].
5. *Uute andmete lühike eluiga*. Loodud fail kustutatakse suure tõenäosusega lühikese aja pärast; faili kirjutatud baidid kirjutatakse suure tõenäosusega lühikese aja pärast üle. 80% failidest kirjutatakse täielikult üle või kustutatakse vähem kui 3 minutit pärast nende loomist [Oust85, Baker91]. Kui modifitseeritakse faili mingeid

blokke, siis keskmiselt 25% sellistest blokkidest kirjutatakse üle või kustutatakse vähem kui 30 sekundi pärast ning keskmiselt 50% neist vähem kui 5 minuti pärast [Oust85].

6. *Väikeste failide kasutamine*. Rohkem kui 75% kõigist failide sulgemistest on sellised, kus fail on sulgemise hetkel alla 10Kb suurune [Oust85, Baker91]. Samas on kasutatavate failide keskmine suurus aasta-aastalt kasvanud. John Ousterhout leidis 1985. aastal, et suurimad kasutatavad failid on umbes 1Mb suurused [Oust85], 1991. aastal läbiviidud kordusuuring tuvastas, et tuleb ette juba 20Mb suuruste failide kasutamist [Baker91].
7. *Failide lühike kasutamisaeg*. 75% juhtudest avatakse fail vähem kui 0,5 sekundiks ning 90% juhtudest vähem kui 10 sekundiks [Oust85, Nelson87, Baker91].
8. *Failide harv üheaegne jagamine*. Olukorrad, kus mitu klientprotsessi on faili üheaegselt avanud, esinevad väga harva. James Thompsoni mõõtmiste kohaselt avatakse ainult 3,2% juhtudest selline fail, mis on antud hetkel juba teise klientprotsessi poolt avatud [Thom87]. Ka Chris Kent on leidnud, et sellise avamise tõenäosus on väiksem kui 4%. Selline üheaegne jagamine, kus vähemalt üks klientprotsess on faili avanud kirjutamiseks, ehk *üheaegne kirjutav jagamine* (*concurrent write sharing*) esineb veelgi harvemini. Ainult 2,2% avamistest on sellised, kus avatakse üheaegselt kirjutavalt jagatud fail või kus avamise tulemusena tekib üheaegne kirjutav jagamine [Thom87].
9. *Failide sage sõlmedevaheline jagamine*. Võrdlemisi tihti esinevad olukorrad, kus üht ja sama faili kasutavad erinevates sõlmedes töötavad klientprotsessid (vastavalt eelmisele failikasutusmallile enamasti siiski mitte üheaegselt). Matthew Blaze on leidnud, et rohkem kui 60% failide lugemiseks avamistest on sellised, kus avatavat faili on kasutanud ka mõnes teises sõlmes töötav (või töötanud) klientprotsess [Blaze93]. Samas modifitseeritakse sõlmede vahel jagatavaid faile väga harva. Matthew Blaze on leidnud, et vähem kui 5% failide kirjutamiseks avamistest on sellised, kus avatavat faili on kasutanud kahes või enamas sõlmes töötavad (või töötanud) klientprotsessid [Blaze93].

Ülaltoodud tulemused puudutavad kahjuks ainult faile; pole teada ühtegi uurimust, milles oleks püütud üksikasjalikult kindlaks teha kataloogide või failiatribuutide kasutamismalle. On siiski üldiselt teada, et ka kataloogide ja failiatribuutide kasutamine

on ajaliselt lokaalne - kui viidatakse neis sisalduvatele andmetele, siis suure tõenäosusega viidatakse neile andmetele lühikese aja pärast uuesti [Sat93]. Siiani on hajusate failisüsteemide vahemälude disainimisel arvestatud siiski eestkätt failikasutusmalle.

3.1.2 Kliendi ja serveri vahemälu

Kliendi vahemälu koosneb tavaliselt kolmest erinevast osast - *andmevahemälust* (*data cache*), *nimevahemälust* (*name cache*) ning *staatusvahemälust* (*status cache*). Andmevahemälus hoitakse failide sisu, nimevahemälus kataloogide sisu ning staatusvahemälus failiatribuute. Mõnedes hajusates failisüsteemides (näiteks AFS) eraldi nimevahemälu puudub ning kataloogide sisu hoitakse andmevahemälus. Sprite on ainuke laiemalt tuntud hajus failisüsteem, kus kliendi vahemälu koosneb ainult andmevahemälust ning nime- ja staatusvahemälu puuduvad. Sprite'i autorid hindasid 1992. aastal nime- ja staatusvahemälude kasutamise otstarbekust. Nad leidsid, et ainult 10 kataloogifaili mahutava nimevahemälu kasutamisel on 82% failinimedest sellised, mille interpreteerimise käigus ei pea Sprite'i kliendimoodul kordagi serverprotsessi poole pöörduma. Ainult 10 faili atribuute mahutav staatusvahemälu vähendaks atribuutide lugemise eesmärgil toimuvaid serverprotsessi poole pöördumisi rohkem kui 4 korda [Shir92].

Andme-, nime- ja staatusvahemälu iseloomustavad parameetrid on järgmised:

- **Vahemälu suurus.**
- **Vahemälu asukoht.** Andme- ja nimevahemälu võivad paikneda klientarvuti operatiivmälus või kettal, staatusvahemälu asub tavaliselt operatiivmälus.
- **Vahemälu element.** Andmevahemälu elementideks võivad olla failid või failiblokid. Nimevahemälu elementideks võivad olla üksikud kataloogifaili kirjed, kataloogifaili blokid või terved kataloogifailid. Staatusvahemälu elementideks on failiatribuute sisaldavad kirjed.
- **Andmete vahemällu paigutamise strateegia** ehk **paigutamisstrateegia** (*placement policy*). See strateegia määrab, millal vahemällu milliseid uusi elemente paigutatakse.

- **Andmete vahemälust väljajätmise strateegia** ehk **asendamisstrateegia** (*replacement policy*). See strateegia määrab, milliseid elemente pärast vahemälu täissaamist sealt välja jätta, et uutele elementidele ruumi teha.
- **Kirjutamisstrateegia** (*writing policy*). See strateegia määrab, kui kaua andmetesse tehtud muudatusi enne serverprotsessile saatmist vahemälus hoitakse.
- **Vahemälu valideerimise strateegia** ehk **valideerimisstrateegia** (*validation policy*). See strateegia määrab, kuidas kindlaks teha, millised vahemälu elemendid on mittekooskõlaliseks muutunud.

Enamus operatsioonisüsteeme eraldab ketta koormuse vähendamiseks osa operatiivmälust *kettavahemälu* (*disk cache*) jaoks, kus hoitakse hiljuti kasutatud kettablokke. Hajusates failisüsteemides kasutatakse serveri vahemäluna tavaliselt serverarvuti operatsioonisüsteemi kettavahemälu.

Serveri vahemälu parameetrid on järgmised:

- **Vahemälu suurus.**
- **Vahemälu elemendi (kettabloki) suurus.**
- **Paigutamisstrateegia.** See strateegia määrab, millal serveri vahemällu milliseid uusi blokke paigutatakse.
- **Asendamisstrateegia.** See strateegia määrab, milliseid blokke pärast serveri vahemälu täissaamist sealt välja jätta, et uutele blokkidele ruumi teha.
- **Kirjutamisstrateegia.** See strateegia määrab, kui kaua modifitseeritud blokke enne serverarvuti kettale kirjutamist serveri vahemälus hoitakse.

Hajusa failisüsteemi vahemälude disainimine seisneb kliendi ja serveri vahemälude parameetrite fikseerimises ning *vahemälustrateegia* (*caching policy*) valikus. Vahemälustrateegia määrab, milliseid vahemälusid andmete hankimisel kasutatakse ning millises järjekorras neid kasutatakse. Levinuim on klient-server mudelil põhinevates hajusates failisüsteemides kasutatav vahemälustrateegia, kus andmeid püütakse kõigepealt kliendi vahemälust hankida ning kui see ebaõnnestub, siis serveri vahemälust. Kui ka seal andmeid ei ole, pöördutakse nende omandamiseks lõpuks serverarvuti ketta[7] poole.

[7] Mõnedes töödes [Dahlin94b] vaadeldakse serverarvuti ketast samuti vahemäluna. Klient-server mudelil põhinevas hajusas failisüsteemis sisaldab see vahemälu alati kõiki vajalikke andmeid.

Leiduvad ka teistsugused vahemälustrateegiad, kus peale *lokaalse* kliendi vahemälu otsitakse vajalikke andmeid ka *mõne teise klientarvuti juures paiknevast* kliendi vahemälust [Blaze93, Dahlin94a, Dahlin94b, Wang93]. Selliste vahemälustrateegiate kasutamine tähendab klient-server mudelist eemaldumist, sest hajussüsteemi ressursid (kauged failid) on hajutatud paljude süsteemi sõlmede (klientarvutite) juurde ning peale serverprotsesside võivad klientprotsesse teenindada ka teiste klientarvutite juures asuvad kliendimoodulid. Taolisi alternatiivseid vahemälustrateegiaid vaadeldakse lähemalt selle peatüki viimases punktis; kõigis selle peatüki ülejäänud punktides on eeldatud, et hajus failisüsteem on üles ehitatud klient-server mudelile.

3.1.3 Vahemälude disaini efektiivsuse hindamine

Vahemälusid kasutatakse hajusa failisüsteemi skaleeritavuse ja klientprotsesside töökiiruse suurendamiseks. Mil määral neid eesmärke saavutada õnnestub, sõltub vahemälude disainist. Mida skaleeritavam on hajus failisüsteem ning mida suurem on klientprotsesside töökiirus, seda efektiivsemaks vahemälude disaini loetakse. Et süsteem oleks võimalikult skaleeritav, peab selles olema võimalikult vähe süsteemi laienemist takistavaid pudelikaelu. Klient-server mudelil põhinevas hajusas failisüsteemis on põhilisteks pudelikaelteks serverarvuti ja sellega ühendatud kohtvõrk (vt. lk. 13). Mida vähem koormatud mingi vahemälude disaini korral serverarvuti ja sellega ühendatud kohtvõrk on, seda skaleeritavam on hajus failisüsteem, sest süsteem võib rohkem laieneda, ilma et serverarvuti ja kohtvõrk veel pudelikaelteks muutuksid. Mida vähem koormatud nad mingi vahemälude disaini korral on, seda efektiivsem on järelikult ka vahemälude disain.

Vahemälude disaini efektiivsust hinnatakse hajusa failisüsteemi prototüübi testimisega või kui prototüüp puudub, siis simulatsioonide abil.

Hajusa failisüsteemi prototüübi testimiseks luuakse prototüübi installatsioon ja käivitatakse sellesse kuuluvates klientarvutites testprogrammid ehk *testid* (*benchmarks*), mis kasutavad kaugeid faile ja jäljendavad sellega klientprotsesside tööd. Testid on nii kirjutatud, et kaugeid faile kasutataks intensiivselt ja asetataks sellega hajusa failisüsteemi prototüübi installatsioonile suur koormus. Tuntuim selline test on

Andrew test [Howard88]. Kui testimise eesmärgiks on hinnata vahemälude disaini efektiivsust hajusa failisüsteemi skaleeritavuse suurendamisel, mõõdetakse testimise käigus tavaliselt serverarvutiga ühendatud kohtvõrgu ning serverarvuti protsessori ja ketta koormust. Mida vähem need testimise ajal koormatud on, seda efektiivsemaks (süsteemi skaleeritavuse suurendamise seisukohalt) vahemälude disaini loetakse.

Kui testimise eesmärgiks on hinnata vahemälude disaini efektiivsust klientprotsesside töökiiruse suurendamisel, mõõdetakse testprogrammide täitmise keskmist või maksimaalset aega, mis annab teatava ettekujutuse sellest, kui kiiresti tavalised klientprotsessid antud vahemälude disaini korral töötaksid. Mida lühem mõõdetud aeg on, seda efektiivsemaks (klientprotsesside töökiiruse suurendamise seisukohalt) vahemälude disaini loetakse.

Kuigi testimise eeliseks on saadud tulemuste täpsus, nõuab see hajusa failisüsteemi prototüübi olemasolu. Prototüübi puudumisel kasutatakse vahemälude disaini efektiivsuse hindamiseks *simulatsioone* (*simulations*) - modelleeritakse mingi programmi abil hajusa failisüsteemi tööd antud vahemälude disaini korral. Simulatsiooni läbiviimiseks on vaja lähteandmeid, mis kajastaksid klientprotsesside käitumist tegelikus hajusas failisüsteemis. Lähteandmed kogutakse tavaliselt mõnest eksisteerivast hajusast failisüsteemist, modifitseerides iga klientarvuti operatsioonisüsteemi tuuma. Kui klientprotsess pöördub failitöötluse süsteemifunktsiooni poole, salvestab klientarvuti operatsioonisüsteemi tuum kõik, mis antud pöördumist puudutab (süsteemifunktsiooni nimi, selle parameetrid, pöördumise aeg ja täitmisele kulunud aeg, klientprotsessi identifikaator ja omanik jms.).

Kui simulatsiooni läbiviimise eesmärgiks on hinnata vahemälude disaini efektiivsust hajusa failisüsteemi skaleeritavuse suurendamisel, mõõdetakse simulatsiooni käigus tavaliselt kujuteldava serverarvuti protsessori ja ketta ning serverarvutiga ühendatud kujuteldava kohtvõrgu koormust. Mida vähem need hajusa failisüsteemi töö simuleerimise ajal koormatud on, seda efektiivsemaks (süsteemi skaleeritavuse suurendamise seisukohalt) vahemälude disaini loetakse.

Kui simulatsiooni läbiviimise eesmärgiks on hinnata vahemälude disaini efektiivsust klientprotsesside töökiiruse suurendamisel, mõõdetakse kujuteldavate klientprotsesside töötamise aegu, mis annab teatava ettekujutuse sellest, kui kiiresti tegelikud klientprotsessid antud vahemälude disaini korral töötaksid. Mida lühemad

kujuteldavate klientprotsesside töötamise ajad on, seda efektiivsemaks (klientprotsesside töökiiruse suurendamise seisukohalt) vahemälude disaini loetakse.

Et simulatsioonide abil klientprotsesside töökiirust või serverarvuti ja sellega ühendatud kohtvõrgu koormust hinnata, tuleb kohtvõrgu läbilaskevõime ning klient- ja serverarvutite riistvara jõudluse kohta teatud eeldusi teha. Vahel hinnatakse serverarvuti ja kohtvõrgu koormust kliendimooduli ja serverprotsessi vahel võrgu kaudu edastatud teadete *arvu* järgi, jättes mainitud eeldused tegemata. Sellise lähenemise korral jaotatakse teated liikidesse ning eeldatakse, et kõik serverprotsessi poolt vastuvõetud või saadetud üht liiki teated koormavad serverarvutiga ühendatud kohtvõrku ning serverarvuti protsessorit ja ketast ühepalju.

3.1.4 Vahemälu vea- ja tabamustegur

Vahemälude disaini efektiivsust hinnatakse tihti hajusa failisüsteemi vahemälude *veategurite* (*miss rates*) ja *tabamustegurite* (*hit rates*) järgi. Vahemälu veategur on defineeritud suhtena, mille lugejaks on antud vahemälu korral toimunud *vahemäluvigade* (*cache misses*) arv ja nimetajaks antud vahemälu kasutamise kordade arv. Vahemäluveaks nimetatakse sündmust, kus vahemälu kasutamisel viidatakse andmetele, mida ei sisalda ükski vahemälu element. Vahemälu tabamustegur on defineeritud avaldisena 1 - veategur.

Kliendi vahemälu osade vea- ja tabamustegur leitakse järgmiselt. Kui klientprotsess pöördub Tabel 3.1-s toodud süsteemifunktsiooni[8] poole, püüab kliendimoodul süsteemifunktsiooni täitmiseks vajalikke andmeid vastavast vahemälust leida. Oletame, et kui kõik vajalikud andmed oleksid vahemälus, sisalduksid nad vahemälu n erinevas elemendis. Kui tegelikult õnnestub kliendimoodulil vahemälust leida m vajalikke andmeid sisaldavat elementi ($m \leq n$), loetakse, et antud vahemälu kasutati n korral ning toimus $n - m$ vahemäluviga. Nimevahemälu puhul arvestatakse vahemäluvigade ja vahemälu kasutamiskordade loendamisel ka kliendimooduli tegevust, kui see püüab failinimede interpreteerimisel selleks vajalikku infot nimevahemälust leida.

[8] Edaspidi on konkreetsuse mõttes eeldatud, et klientprotsessid kasutavad kaugeid faile läbi BSD UNIXi failitöötluse süsteemifunktsioonide.

Vahemälu tüüp	Süsteemi-funktsioon	Süsteemifunktsiooni täitmiseks vajalikud andmed
Andmevahemälu (elementideks on failiblokid)	`read, write`	Failiblokid, mida süsteemifunktsiooni `read` abil lugeda või süsteemifunktsiooni `write` abil modifitseerida kavatsetakse
Andmevahemälu (elementideks on failid)	`open`	Fail, mida süsteemifunktsiooni `open` abil avada kavatsetakse
Nimevahemälu (elementideks on kataloogifaili kirjed)	`readdir`	Kataloogifaili kirje, mida süsteemifunktsiooni `readdir` abil lugeda kavatsetakse
Nimevahemälu (elementideks on kataloogifaili blokid)	`readdir`	Kataloogifaili blokk, kuhu kuuluvat kirjet süsteemifunktsiooni `readdir` abil lugeda kavatsetakse
Nimevahemälu (elementideks on kataloogifailid)	`opendir`	Kataloogifail, mida süsteemifunktsiooni `opendir` abil avada kavatsetakse
Staatusvahemälu	`stat`	Failiatribuudid, mida süsteemifunktsiooni `stat` abil lugeda kavatsetakse

Tabel 3.1 Kliendi vahemälu osade vea- ja tabamusteguri leidmine.

Serveri vahemälu vea- ja tabamustegur leitakse järgmiselt. Oletame, et mingi kaugprotseduuri täitmisel oleks serveri vahemälu puudumisel toimunud *n* bloki kettalt lugemise ja bloki kettale kirjutamise operatsiooni. Kui tegelikult asus vajalik kettablokk *m* operatsiooni puhul serveri vahemälus ($m \leq n$), loetakse, et serveri vahemälu kasutati *n* korral ning toimus $n - m$ vahemäluviga.

Kui vahemälu kasutamise kordi ja vahemäluvigu on pikema aja jooksul loendatud, näitab vahemälu tabamustegur, kui suur osa vajalikest andmetest tavaliselt vahemälus asub, ning veategur, kui suur osa vajalikest andmetest sealt tavaliselt puudub. Mida kõrgemad on kliendi vahemälu osade tabamustegurid ja madalamad nende veategurid, seda vähem kliendimoodul serverprotsessi poole pöördub; seda väiksem on järelikult serverarvuti ja sellega ühendatud kohtvõrgu koormus ning seda suurem on klientprotsessi töökiirus. Mida kõrgem on serverarvuti vahemälu tabamustegur ja madalam selle veategur, seda väiksem on serverarvuti ketta ja protsessori koormus ning seda suurem on klientprotsesside töökiirus (sest andmete lugemine serveri vahemälust toimub kiiremini kui serverarvuti kettalt).

Mida kõrgemad on hajusa failisüsteemi vahemälude tabamustegurid ning mida madalamad vahemälude veategurid, seda efektiivsemaks vahemälude disaini loetakse.

3.1.5 Kooskõlalisuse semantika

Operatsioonisüsteemi või hajusa failisüsteemi *kooskõlalisuse semantika* (*consistency semantics*) näitab, millal mingi protsessi poolt faili tehtud muudatused teistele protsessidele nähtavaks saavad. UNIXi (ja paljude teiste operatsioonisüsteemide) kooskõlalisuse semantika on järgmine:

- kui protsess kirjutab faili, muutuvad faili tehtud muudatused kõigile antud faili avatuna hoidvatele protsessidele nähtavaks vahetult pärast kirjutamisoperatsiooni lõppu.
- kõik protsessid, kes faili pärast kirjutamisoperatsiooni lõppu avavad, näevad samuti tehtud muudatusi.

Praktiliste hajusate failisüsteemide kooskõlalisuse semantika erineb tavaliselt UNIXi kooskõlalisuse semantikast. Näiteks AFSi puhul näeb kirjutamisstrateegia ette, et faili tehtud muudatused saadetakse serverprotsessile alles faili sulgemisel. Kui hajusa failisüsteemi erinevates sõlmedes töötavad klientprotsessid hoiavad faili üheaegselt avatuna ja üks neist faili kirjutab, ei muutu faili kirjutatud baidid seetõttu teistele klientprotsessidele kohe nähtavaks. Kuna aga failide üheaegne jagamine esineb harva (vt. alapunkt 3.1.1), on paljud eriteadlased seisukohal, et hajusa failisüsteemi kooskõlalisuse semantika erinevus UNIXi kooskõlalisuse semantikast on teisejärguline probleem [Coul94, Howard88].

Ainus laiemalt tuntud hajus failisüsteem, kus UNIXi kooskõlalisuse semantikat täpselt järgitakse, on Sprite. Kui fail muutub üheaegselt kirjutavalt jagatavaks, keelab serverprotsess kliendimoodulitel selle faili blokkide vahemälus hoidmise. Sellisest failist lugemiseks või sinna kirjutamiseks tuleb alati otse serverprotsessi poole pöörduda [Nelson87].

3.2 Paigutamis- ja asendamisstrateegiad

3.2.1 Lihtsamad kliendi vahemälu paigutamis- ja asendamisstrateegiad

Nii andme-, nime- kui staatusvahemällu paigutatakse uusi andmeid tavaliselt siis, kui neile andmetele viidatakse. See paigutamisstrateegia on küllalt efektiivne (vt. lk. 66), sest ta arvestab failide, kataloogide ja failiatribuutide kasutamise ajalist lokaalsust (vt. alapunkt 3.1.1) - kui failis, kataloogis või failiatribuutides sisalduvatele andmetele viidatakse, siis suure tõenäosusega viidatakse neile lühikese aja pärast uuesti.

Kui andmevahemälu elementideks on failid, paigutab kliendimoodul uue faili vahemällu siis, kui klientprotsess selle süsteemifunktsiooni `open` abil avab ning avatavat faili vahemälus ei ole[9]. Kui kliendimoodul pärast faili avamist avastab, et fail on mittekooskõlaliseks muutunud, siis enamikes praktilistes hajusates failisüsteemides kliendimoodul faili uuesti vahemällu ei paiguta (sellist lähenemist kasutavad näiteks AFS ja Coda).

Kui andmevahemälu elementideks on blokid, paigutatakse sinna uusi blokke järgmistel juhtudel. Kui blokki vahemälus pole ning klientprotsess sellesse kuuluvaid baite süsteemifunktsiooni `read` abil lugeda püüab, pöördub kliendimoodul vajaliku bloki omandamiseks serverprotsessi poole ja paigutab selle ühtlasi vahemällu. Kui klientprotsess vahemälust puuduvat blokki süsteemifunktsiooni `write` abil modifitseerib, paigutab kliendimoodul selle bloki samuti vahemällu. Kui blokki modifitseeritakse *osaliselt*, pöördub kliendimoodul puuduva bloki omandamiseks kõigepealt serverprotsessi poole ja seejärel uuendab juba vahemälus asuvat blokki (muudatuste vahemällu salvestamiseks on vaja teada ka bloki neid baite, mida klientprotsess ei muutnud). Juhul, kui klientprotsess modifitseerib vahemälust puuduvat blokki *täielikult* (s.t. muudab selle kõiki baite) või lisab faili lõppu uue bloki, paigutatakse täielikult modifitseeritud või lisatud blokk vahemällu ilma serverprotsessi poole pöördumata.

Kui andmevahemälu elementideks on blokid, on otstarbekas kasutada *eellugemist* (*prefetching*) - kliendimoodul paigutab *ennetavalt* vahemällu neid blokke,

[9] Edaspidi loetakse lihtsuse mõttes element vahemälust puuduvaks ka juhul, kui element on tegelikult küll vahemälus, kuid on kehtetuks tunnistatud.

mida klientprotsessid lähemas tulevikus tõenäoliselt kasutada püüavad. Eellugemine võimaldab tõsta klientprotsesside töökiirust, sest blokile viitamise hetkel on see juba vahemälus. *Nõudmisel toimuva eellugemise* (*on-demand-prefetching*) korral toimub eellugemine ainult siis, kui klientprotsess püüab kasutada vahemälust puuduvat blokki, mille omandamiseks peab kliendimoodul serverprotsessi poole pöörduma. Koos hetkel vajaliku blokiga loeb kliendimoodul vahemällu ka neid blokke, millele lähemas tulevikus tõenäoliselt viidatakse. Kliendimoodulil ei ole lubatud pöörduda serverprotsessi poole *ainult* eellugemise eesmärgil.

Kuna klientprotsessid kasutavad faile enamasti järjestikuselt (vt. alapunkt 3.1.1), siis on enamik eellugemise algoritme sellised, mille kohaselt faili *i*-ndale blokile viitamisel loeb kliendimoodul ennetavalt vahemällu faili (*i+1*)-nda bloki. Tuntuim nõudmisel toimuva eellugemise algoritm on *UL* (*UNIX*[10] *lookahead*) - kui klientprotsess viitab järjest faili (*i-1*)-ndale ja *i*-ndale blokile ning *i*-ndat blokki vahemälus pole, siis loeb kliendimoodul vahemällu nii faili *i*-nda kui (*i+1*)-nda bloki (kui (*i+1*)-s blokk on vahemälus, siis eellugemist loomulikult ei toimu ning kliendimoodul loeb ainult *i*-nda bloki) [Thom87].

Tuntuim hariliku eellugemise algoritm on *OBL* (*one-block lookahead*) - kui klientprotsess avab faili ja faili esimest blokki vahemälus pole, loeb kliendimoodul selle ennetavalt; kui klientprotsess viitab faili *i*-ndale blokile ja (*i+1*)-ndat blokki vahemälus pole, loeb kliendimoodul vahemällu nii faili *i*-nda kui (*i+1*)-nda bloki (kui *i*-s blokk on vahemälus, loeb kliendimoodul ainult (*i+1*)-nda bloki) [Thom87].

Nõudmisel toimuva eellugemise korral on eksimuse hind väiksem - kui kliendimoodul loeb ennetavalt sellise bloki, mida klientprotsess tulevikus tegelikult ei kasuta, ei ole nõue serverprotsessile siiski asjata saadetud, sest alati loetakse ka klientprotsessile antud hetkel vajalik blokk. Kui kliendimoodul aga blokkide ennetaval lugemisel ei eksi, suurendab harilik eellugemine klientprotsesside töökiirust rohkem kui nõudmisel toimuv eellugemine, sest blokkide ennetavaks lugemiseks kitsendusi ei ole.

Kui nimevahemälu elementideks on kataloogifailid, paigutab kliendimoodul uue kataloogifaili vahemällu siis, kui klientprotsess selle süsteemifunktsiooni `opendir` abil avab ning kataloogifaili vahemälus ei ole. Kui kliendimoodul pärast kataloogifaili avamist avastab, et fail on mittekooskõlaliseks muutunud, siis enamikes praktilistes

[10] Seda eellugemise algoritmi kasutavad kettavahemälu puhul mõned UNIXi versioonid.

hajusates failisüsteemides kliendimoodul kataloogifaili uuesti vahemällu ei paiguta (sellist lähenemist kasutavad näiteks AFS ja Coda).

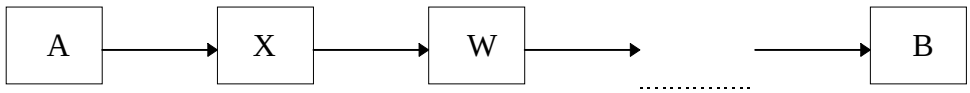
Kui nimevahemälu elementideks on kataloogifailide blokid, paigutab kliendimoodul uue bloki vahemällu siis, kui klientprotsess süsteemifunktsiooni `readdir` abil mõnda blokki kuuluvat kirjet lugeda püüab. Kui nimevahemälu elementideks on kataloogifailide kirjed, paigutab kliendimoodul uue kirje vahemällu siis, kui klientprotsess seda kirjet süsteemifunktsiooni `readdir` abil lugeda püüab.

Tavaliselt paigutatakse nimevahemällu uusi elemente ka siis, kui kliendimoodul loeb failide tekstiliste nimede interpreteerimise käigus kataloogifaile (näiteks AFSi korral operatsiooni `Fetch` abil).

Staatusvahemällu paigutatakse uusi elemente kõigepealt siis, kui klientprotsess viitab süsteemifunktsiooni `stat` abil atribuutidele, mida parajasti vahemälus ei ole. Uusi elemente paigutatakse staatusvahemällu ka siis, kui kliendimoodul need mõne tegevuse käigus ilma klientprotsessi otsese nõudmiseta omandab (NFSi korral näiteks failibloki kooskõlalisuse kontrollimisel operatsiooni `getattr` abil või failibloki lugemisel operatsiooni `read` abil).

Nii andme-, nime- kui staatusvahemälu puhul kasutatakse tavaliselt *LRU-asendamisstrateegiat* (*Least Recently Used replacement policy*) - kui vahemälu on täis ning paigutamisstrateegia kohaselt tuleb vahemällu uus element paigutada, siis uue elemendi jaoks ruumi tegemiseks jäetakse vahemälust välja selline element, mille viimane kasutamine toimus kõige kaugemas minevikus. LRU-asendamisstrateegia on küllalt efektiivne (vt. lk. 66), sest ta arvestab failide, kataloogide ja failiatribuutide kasutamise ajalist lokaalsust. Jättes vahemälust välja viimati kõige kauem aega tagasi kasutatud elemendi, jäävad vahemällu seega kõik elemendid, mida on kasutatud lähemas minevikus ning mille korral on tõenäoline, et neid lähemas tulevikus uuesti kasutatakse. LRU-asendamisstrateegia realiseerimiseks peab igal vahemälu elemendil olema unikaalne nimi. Kliendimoodul peab kõigi vahemälus sisalduvate elementide kohta *LRU-nimekirja* (*LRU-list*), mis koosneb elementide nimedest ning kus iga elemendi nimi esineb täpselt ühel korral (vt. Joonis 3.1). Kui elemendile viidatakse, paigutab kliendimoodul elemendi nime LRU-nimekirja algusse. Kui vahemälu on täis, siis uuele elemendile ruumi tegemiseks jäetakse vahemälust ja LRU-nimekirjast välja element, mille nimi on nimekirja lõpus, ning uue elemendi nimi paigutatakse nimekirja

algusse. Uuele infole ruumitegemiseks saab vahemälust ja LRU-nimekirjast välja jätta ka kustutatud või mittekooskõlaliseks muutunud ja kehtetuks tunnistatud elemente.



Joonis 3.1 LRU-nimekiri (elemendile B on viidatud kõige lähemas minevikus).

Peaaegu kõik praktilised hajusad failisüsteemid kasutavad kliendi vahemälu puhul LRU-asendamisstrateegiat ning paigutamisstrateegiat, mille kohaselt paigutatakse vahemällu uusi andmeid siis, kui neile andmetele viidatakse. Hajusates failisüsteemides, kus neid strateegiaid kasutatakse ning mille vahemälude disainid muus osas üksteisest tunduvalt erinevad, on andme-, nime- ja staatusvahemälude tabamustegurid küllalt kõrged, ületades peaaegu alati 50% piiri [Baker91, Blaze93, Howard88, Spas93]. Näiteks 1993. aastal läbiviidud uuring tuvastas, et AFS-3 ülemaailmse ulatusega installatsioonis (vt. lk. 40) on andme- ja nimevahemälu keskmine tabamustegur 98% ja staatusvahemälu keskmine tabamustegur 96% [Spas93].

Asendamisstrateegia efektiivsuse hindamiseks on paljudes töödes kasutatud võtet, kus asendamisstrateegiat võrreldakse MIN-asendamisstrateegiaga. Kui vahemälu on täis ja on vaja uuele elemendile ruumi teha, jäetakse MIN-asendamisstrateegia kohaselt vahemälust välja selline element, mille järgmine kasutamine toimub kõige kaugemas tulevikus. Praktikas seda asendamisstrateegiat kasutada ei saa, kuna ta nõuab tuleviku etteteadmist.

Kui kõigil vahemäluvigadel on ühesugune hind ning vahemällu paigutatakse uusi andmeid siis, kui neile andmetele viidatakse, on MIN-asendamisstrateegia vahemäluvigade kogumaksumuse vähendamise seisukohalt optimaalne [Blaze93]. MIN-asendamisstrateegia kasutamisel toimub vähim arv vahemäluvigu, sest lugedes iga vahemäluvea hinnaks 1 ühiku, on vahemäluvigade kogumaksumus võrdne toimunud vahemäluvigade arvuga. See tähendab, et MIN-asendamisstrateegia kasutamisel saavutatakse alati vähemalt sama madal veategur ja sama kõrge tabamustegur kui suvalise teise asendamisstrateegia kasutamisel.

3.2.2 Kahetasemeline asendamisstrateegia

Alapunktis 3.2.1 vaadeldud LRU-asendamisstrateegia on *ühetasemeline* - kliendimoodul otsustab ilma klientprotsessidega konsulteerimata, milline element vahemälust välja jätta. Samas oleks tunduvalt otstarbekam, kui kliendimoodul rakendaks *kahetasemelist asendamisstrateegiat* (*two-level replacement policy*) - võtaks selliseid otsuseid vastu koos klientprotsessidega. Kuna klientprotsessid omavad sageli juba oma töö algul täpset infot oma failikasutusmallide kohta, võib vahemälu tabamustegur olla kahetasemelise asendamisstrateegia rakendamisel kõrgem kui ühetasemelise asendamisstrateegia rakendamisel. Esimese (ja siiani ainsa) kahetasemelise asendamisstrateegia töötasid 1994. aastal välja Cao, Felten ja Li [Cao94a]. See strateegia on mõeldud andmevahemälu jaoks, mille elementideks on blokid.

Kui vahemälus asuv blokk B kuulub faili F, siis mingil hetkel *t* loetakse bloki B *omanikuks* seda klientprotsessi, kes faili F sel hetkel *ainsana* avatuna hoiab. Kui fail F pole sel hetkel avatud või faili F hoiavad avatuna mitu klientprotsessi, loetakse blokk B omanikuta blokiks.

Kui vahemälu on täis ning on vaja uuele blokile ruumi teha, rakendab kliendimoodul järgmist *jaotusstrateegiat* (*allocation policy*). Kõigepealt valib kliendimoodul *kandidaatbloki* (*candidate block*) ehk *kandidaadi*, mis tema arvates tuleks vahemälust välja jätta. Kui kandidaadil on sel hetkel omanik olemas, pöördub kliendimoodul seejärel lõpliku otsuse langetamiseks omanikuks oleva klientprotsessi poole. Kandidaadi omanik võib *privaatse asendamisstrateegia* põhjal valida kandidaadi asemel vahemälust väljajätmiseks mõne teise talle kuuluva *alternatiivse bloki* (*alternative block*). Kliendimoodul austab kandidaadi omaniku otsust ja asendab alternatiivse bloki uue blokiga. Kui kandidaadi omanik keeldub koostööst või kandidaadil omanik puudub, asendab kliendimoodul kandidaadi.

Ülaltoodud kliendimooduli poolt rakendatud strateegiat nimetatakse jaotusstrateegiaks, sest kui ühele omanikule kuuluv blokk asendatakse teisele omanikule kuuluvaga, vähendatakse sellega tegelikult ühele klientprotsessile kuuluvat vahemälu osa ühe bloki võrra ja antakse vabanenud ruum teisele klientprotsessile.

Klientprotsessi otsust nimetatakse *targaks* (*wise*), kui klientprotsessi poolt asendamiseks pakutud alternatiivsele blokile viidatakse hiljem kui kliendimooduli poolt pakutud kandidaadile, ning *rumalaks* (*foolish*), kui kandidaadile viidatakse hiljem kui alternatiivsele blokile. Kui klientprotsess nõustub kliendimooduli valikuga või keeldub koostööst, nimetatakse tema otsust *neutraalseks* (*neutral*)[11]. Klientprotsessi, kes langetab alati neutraalseid otsuseid, nimetatakse *hajameelseks* (*oblivious*).

Cao, Felten ja Li formuleerisid ka tingimused, mida hea jaotusstrateegia peab rahuldama. Nende arvates peaks jaotusstrateegia tagama, et iga klientprotsess kannataks väiksema arvu vahemäluvigade läbi kui ühetasemelise asendamisstrateegia kasutamisel. Öeldakse, et klientprotsess *kannatab vahemäluvea läbi* (*suffers from cache miss*), kui klientprotsess viitab vahemälust puuduvale blokile. Mida väiksema arvu vahemäluvigade läbi klientprotsess kannatab, seda suurem on ta töökiirus, sest uue bloki vahemällu paigutamine on enamasti aeganõudev operatsioon. Kuna andmevahemälu korral kasutatakse tavaliselt ühetasemelist LRU-asendamisstrateegiat (edaspidi tähistatud 1-LRU), siis kasutasid Cao, Felten ja Li tingimuste formuleerimisel seda konkreetset asendamisstrateegiat:

1. Hajameelne klientprotsess ei tohi kannatada suurema arvu vahemäluvigade läbi kui ta kannataks 1-LRU-asendamisstrateegia kasutamisel.
2. Hoolimata klientprotsessi rumalast otsusest ei tohi ükski teine klientprotsess kannatada suurema arvu vahemäluvigade läbi kui ta kannataks 1-LRU-asendamisstrateegia kasutamisel.
3. Klientprotsessi tark otsus ei tohi kaasa tuua olukorda, kus mõni klientprotsess (kaasa arvatud ta ise) kannatab suurema arvu vahemäluvigade läbi kui ta kannataks 1-LRU-asendamisstrateegia kasutamisel.

Püüdes leida neile tingimustele vastavat jaotusstrateegiat, võtsid Cao, Felten ja Li otsingutel aluseks ALLOC-LRU-jaotusstrateegia, mis on analoogiline 1-LRU-asendamisstrateegiaga. Kliendimoodul peab vahemälus asuvate blokkide kohta LRU-nimekirja ja valib asendamisel kandidaadiks bloki, mille nimi on LRU-nimekirja lõpus. Kui kõik klientprotsessid on hajameelsed, pole nende seisukohalt mingit vahet, kas kliendimoodul rakendab 1-LRU-asendamisstrateegiat või kahetasemelist

[11] Cao, Felten ja Li on otsuseid nii klassifitseerinud nähtavasti seepärast, et võrreldes kliendimooduli ettepanekuga on tark otsus MIN-asendamisstrateegiale “lähemal” ja rumal otsus sellest “kaugemal”.

asendamisstrateegiat, kus jaotusstrateegiaks on ALLOC-LRU. Sellisel juhul on kõik kolm tingimust täidetud (2. ja 3. tingimuse täidetust pole vaja kontrollida, sest klientprotsessid teevad alati neutraalseid otsuseid).

Kui kõik klientprotsessid pole hajameelsed, ei vasta ALLOC-LRU-jaotusstrateegia enam esitatud tingimustele. Vaatleme järgmist näidet (vt. Tabel 3.2) - klientprotsessid P ja Q kasutavad 4 bloki suurust vahemälu, kus blokid A ja B kuuluvad klientprotsessile P ning blokid W ja X kuuluvad klientprotsessile Q. Algul viitab klientprotsess Q blokile Y, mis toob kaasa asenduse. Kliendimoodul valib kandidaadiks bloki A ja pöördub klientprotsessi P kui kandidaadi omaniku poole. P otsustab, et bloki B asendamine on otstarbekam. See on tark otsus, sest tulevikus viidatakse blokile A varem kui blokile B. Kliendimoodul asendab bloki B blokiga Y. Seejärel viitab klientprotsess Q blokile Z ning kliendimoodul valib jälle kandidaadiks bloki A, sest bloki A nimi on LRU-nimekirja lõpus. Kuna see on ainus blokk, mis klientprotsessile P veel kuulub, ei ole tal võimalik alternatiivset blokki välja pakkuda ning blokk A asendatakse blokiga Z. Järgnevalt viitab klientprotsess P blokkidele A ja B ning kannatab mõlemal juhul vahemäluvea läbi.

P	**Q**	**ALLOC-LRU**		**LRU-S**	
tegevus	**tegevus**	**Vahemälu sisu**	**LRU-nimekiri**	**Vahemälu sisu**	**LRU-nimekiri**
		AWXB	A→W→X→B	AWXB	A→W→X→B
	Y				
		AWXY	A→W→X→Y	AWXY	W→X→A→Y
	Z				
		ZWXY	W→X→Y→Z	AZXY	X→A→Y→Z
A					
		ZAXY	X→Y→Z→A	AZXY	X→Y→Z→A
B					
		ZABY	Y→Z→A→B	AZBY	Y→Z→A→B

Tabel 3.2 ALLOC-LRU ja LRU-S võrdlus.

Selle näite korral leiab ALLOC-LRU-jaotusstrateegia kasutamisel aset neli vahemäluviga (mõlemad klientprotsessid kannatavad kahe vahemäluvea läbi), 1-LRU-asendamisstrateegia korral toimuks vaid kolm vahemäluviga (klientprotsess P

kannataks ühe ja klientprotsess Q kahe vahemäluvea läbi). Seega on rikutud 3. tingimust - klientprotsessi P tark otsus toob tema jaoks kaasa täiendava vahemäluvea.

Cao, Felten ja Li modifitseerisid ALLOC-LRU-jaotusstrateegiat ja nimetasid saadud uue strateegia *LRU-S-jaotusstrateegiaks* (*LRU with Swapping allocation policy*). ALLOC-LRU korral tekivad täiendavad vahemäluvead seetõttu, et alternatiivse bloki asendamisel jääb kandidaadi nimi LRU-nimekirja lõppu. Järgmise asenduse tegemiseks pöördub kliendimoodul jälle sama klientprotsessi poole ning nõuab kandidaadi heakskiitmist või alternatiivse bloki esitamist.

LRU-S korral vahetab kliendimoodul alternatiivse bloki esitamisel kõigepealt LRU-nimekirjas alternatiivse bloki ja kandidaadi nimed (s.t. alternatiivse bloki nimi paigutatakse LRU-nimekirja lõppu) ning alles seejärel asendab alternatiivse bloki uue blokiga (alternatiivse bloki nimi jäetakse loomulikult LRU-nimekirjast välja). Sellega väldib LRU-S-jaotusstrateegia olukorda, kus klientprotsessi tark otsus võiks teda ennast kahjustada. Kui kasutada LRU-S-jaotusstrateegiat, siis ülaltoodud näite korral kannatab klientprotsess P vaid ühe vahemäluvea läbi nagu 1-LRU-asendamisstrateegia puhulgi (vt. Tabel 3.2). Kui ükski klientprotsess ei tee rumalaid otsuseid, siis LRU-S-jaotusstrateegia kasutamisel on vahemälu tabamustegur vähemalt sama kõrge kui 1-LRU-asendamisstrateegia kasutamisel [Cao96].

Kui mingi klientprotsess teeb rumala otsuse, siis kahjuks ei kaitse LRU-S-jaotusstrateegia teisi klientprotsesse selle otsuse eest. Vaatleme järgmist näidet (vt. Tabel 3.3) - klientprotsessid P ja Q jagavad 3 bloki suurust vahemälu, kus blokk A kuulub klientprotsessile P ning blokid X ja Y klientprotsessile Q. Kui klientprotsess Q viitab blokile Z, toob see kaasa asenduse. Kliendimoodul valib kandidaadiks bloki X ja pöördub klientprotsessi Q kui kandidaadi omaniku poole. Klientprotsess Q aga esitab alternatiivse bloki Y, misjärel LRU-nimekirjas vahetatakse omavahel blokkide X ja Y nimed ning blokk Y asendatakse blokiga Z. Klientprotsessi Q otsus on rumal, sest blokile X tulevikus enam ei viidata, küll aga viidatakse blokile Y. Järgnevalt viitab klientprotsess Q blokile Y, mis toob kaasa uue asenduse. Seekord valib kliendimoodul kandidaadiks bloki A ning selle omanik klientprotsess P on sunnitud nõustuma, sest talle ei kuulu rohkem blokke ning seega pole tal võimalik alternatiivset blokki esitada. Seejärel viitab klientprotsess P blokile A, mis toob kaasa uue vahemäluvea ning asenduse.

P tegevus	Q tegevus	LRU-S		LRU-SP	
		Vahemälu sisu	**LRU-nimekiri**	**Vahemälu sisu**	**LRU-nimekiri**
		XAY	X→A→Y	XAY	X→A→Y
	Z				
		XAZ	A→X→Z	XAZ	A→X(Y)→Z
	Y				
		XYZ	X→Z→Y	YAZ	A→Z→Y
A					
		AYZ	Z→Y→A	YAZ	Z→Y→A

Tabel 3.3 LRU-S ja LRU-SP võrdlus.

Selle näite korral leiab LRU-S-jaotusstrateegia kasutamisel aset kolm vahemäluviga (klientprotsess P kannatab ühe ning klientprotsess Q kahe vahemäluvea läbi). 1-LRU-asendamisstrateegia korral oleks klientprotsess Q kannatanud ühe ning klientprotsess P mitte ühegi vahemäluvea läbi. Seega on rikutud 2. tingimust, sest klientprotsessi Q rumal otsus toob klientprotsessi P jaoks kaasa täiendava vahemäluvea. Täiendavad vahemäluvead tekivad sellepärast, et rumala otsuse teinud klientprotsessil on võimalik vahemällu paigutada rohkem blokke kui 1-LRU-asendamisstrateegia korral. See tähendab, et teised klientprotsessid võivad kannatada suurema arvu vahemäluvigade läbi kui nad kannataksid 1-LRU-asendamisstrateegia rakendamisel.

Et takistada rumalaid otsuseid teinud klientprotsessidel teisi klientprotsesse kahjustamast, modifitseerisid Cao, Felten ja Li LRU-S-jaotusstrateegiat ja nimetasid saadud uue strateegia LRU-SP-jaotusstrateegiaks (*LRU with Swapping and Placeholders allocation policy*). Kui kliendimoodul pakub kandidaadina välja bloki A ja klientprotsess esitab alternatiivse bloki B, siis lisaks blokkide A ja B nimede vahetamisele LRU-nimekirjas loob kliendimoodul vahemälust väljajäetava bloki B jaoks *kohahoidja* (*placeholder*), mis viitab blokile A. Kui viidatakse blokile A ja leidub blokile A osutav kohahoidja, siis see kohahoidja kõrvaldatakse. Kui viidatakse blokile B ja selle jaoks leidub blokile A osutav kohahoidja (mis tähendab, et blokile A pole veel viidatud ning otsus bloki B asendamise kohta oli rumal), asendab kliendimoodul bloki A blokiga B ja kõrvaldab kohahoidja. Sellega korrigeerib kliendimoodul klientprotsessi rumalat otsust (leidub LRU-SP-jaotusstrateegia variatsioon, kus kliendimoodul ei asenda blokki A

blokiga B automaatselt, vaid valib bloki A asendamise kandidaadiks, võimaldades rumala otsuse teinud klientprotsessil loobuda mõnest talle vähem vajalikust blokist [Cao94b]).

Kui kliendimoodul valib asendamisel kandidaadiks bloki C, kuid bloki C omanik otsustab asendada bloki A ning leidub blokile A viitav kohahoidja, siis pannakse see kohahoidja viitama blokile C. See välistab rumala otsuse teinud klientprotsessil täiendavate blokkide vahemällu paigutamise võimaluse, mida talle 1-LRU-asendamisstrateegia korral ei antaks.

LRU-SP rakendamisel on klientprotsessid kaitstud teiste klientide rumalate otsuste eest, rumala otsuse teinud klientprotsess kahjustab vaid iseennast. Kui kasutada LRU-SP-jaotusstrateegiat, siis ülaltoodud näite korral ei kannata klientprotsess P enam vahemäluvigade läbi, klientprotsessi Q rumal otsus toob tema jaoks kaasa täiendava vahemäluvea (vt. Tabel 3.3). Saab näidata, et LRU-SP-jaotusstrateegia rahuldab kõiki heale jaotusstrateegiale esitatud tingimusi (formaalse tõestuse võib leida allikast [Cao96]).

Kahetasemelise asendamisstrateegia efektiivsuse hindamiseks viisid Cao, Felten ja Li läbi viis simulatsiooni [Cao94a, Cao96]. Simulatsioonide lähteandmed koguti hajusa failisüsteemi Sprite installatsioonist, kuhu kuulus 40 sõlme. Simulatsioonide läbiviimisel eeldati, et klientprotsessid rakendavad järgmist privaatset asendamisstrateegiat - kui kliendimoodul valib asendamisel kandidaadiks faili F *i*-nda bloki, esitab klientprotsess alternatiivse blokina faili F *j*-nda bloki ($j \geq i$), mille korral vahe $j - i$ oleks suurim. Kuna faile kasutatakse enamasti järjestikuselt ning tihti opereeritakse kogu faili sisuga (vt. alapunkt 3.1.1), on sellise privaatse asendamisstrateegia abil tehtud otsused enamasti targad. Cao, Felten ja Li leidsid, et iga kümnenda andmevahemälu veategur vähenes märgatavalt (parimal juhul rohkem kui 2 korda).

Cao, Felten ja Li on enda väljatöötatud kahetasemelise asendamisstrateegia realiseerinud ka NFSi kliendimooduli jaoks, kus klientarvuti operatsioonisüsteemiks on Ultrix [Cao94b]. Klientprotsesside töökiiruse seisukohalt on ebaotstarbekas, kui kliendimoodul igal asendamisel otsuse langetamiseks mõne klientprotsessi poole pöördub ning klientprotsess igal sellisel juhul privaatset asendamisstrateegiat rakendab. Cao, Felten ja Li on kahetasemelise asendamisstrateegia realiseerinud nii, et

klientprotsessid pöörduvad vajadusel süsteemifunktsiooni `fbehavior` abil kliendimooduli poole ning määravad, millist privaatset asendamisstrateegiat nad kasutada soovivad. Blokkide asendamisel teeb kliendimoodul otsuseid ilma klientprotsesside poole pöördumata, kasutades klientprotsesside poolt ette kirjutatud privaatseid asendamisstrateegiaid. 1994. aastal toetas realisatsioon ainult LRU- ja MRU-asendamisstrateegiaid. *MRU-asendamisstrateegia* (*Most Recently Used replacement policy*) kohaselt jäetakse vahemälust välja element, millele viimati kõige vähem aega tagasi viidati. See privaatne asendamisstrateegia sobib hästi näiteks juhul, kui tegu on blokkide *tsüklilise kasutamisega* - klientprotsess loeb oma töö algul mingis järjekorras erinevaid blokke B_1, ..., B_n, seejärel loeb neid kõiki samas järjekorras uuesti jne.

3.2.3 Serveri vahemälu paigutamis- ja asendamisstrateegiad

Serveri vahemälu aitab vähendada serverarvuti ketta ja protsessori koormust, sest kui kettablokk on vahemälus, pole selle lugemiseks või modifitseerimiseks vaja serverarvuti ketta poole pöörduda. Kuna kettapöördused on palju aeglasemad kui operatiivmälust lugemine või sinna kirjutamine, saab serverprotsess tänu serveri vahemälule klientprotsesse kiiremini teenindada ning klientprotsesside töökiirus kasvab. Kui vahemälus paiknevat blokki modifitseeritakse, määrab tema kettale kirjutamise hetke serveri vahemälu kirjutamisstrateegia.

Analoogiliselt kliendi vahemäluga paigutatakse serveri vahemällu uusi andmeid siis, kui neile andmetele viidatakse. Kui serverprotsessile saabub nõue ning nõudele vastava kaugprotseduuri täitmise käigus on vaja lugeda vahemälust puuduvaid kettablokke, loetakse need blokid serverarvuti kettalt ja paigutatakse ühtlasi serveri vahemällu. Kui nõudele vastava kaugprotseduuri täitmise käigus on vaja modifitseerida vahemälust puuduvaid kettablokke, paigutatakse need blokid samuti serveri vahemällu. Kui vahemälust puuduvat blokki modifitseeritakse osaliselt, loetakse blokk eelnevalt serverarvuti kettalt vahemällu ja seejärel uuendatakse, bloki täielikul modifitseerimisel paigutatakse ta vahemällu ilma serverarvuti ketta poole pöördumata.

Serveri vahemälu asendamisstrateegiana kasutatakse tavaliselt LRU-asendamisstrateegiat.

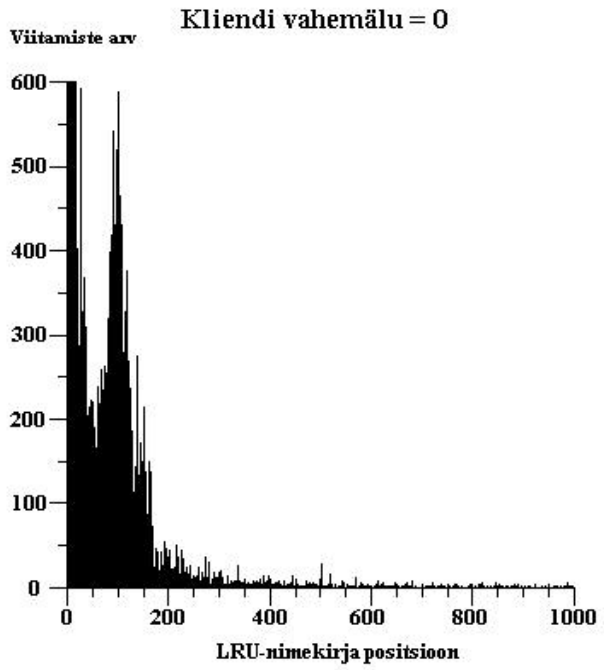
Nagu alapunktis 3.2.1 juba mainitud, on paigutamisstrateegia, kus andmeid paigutatakse vahemällu neile viitamisel, ja LRU-asendamisstrateegia efektiivsed siis, kui andmete kasutamine on ajaliselt lokaalne - kui andmetele viidatakse, siis suure tõenäosusega viidatakse neile lühikese aja pärast uuesti. Neid strateegiaid on serveri vahemälu puhul siiani enim kasutatud, kuid tegelikult ei pruugi nad paljudel juhtudel eriti efektiivsed olla. Kuigi nii failide, kataloogide kui failiatribuutide kasutamine on ajaliselt lokaalne, pöördub kliendimoodul serverprotsessi poole ainult siis, kui klientprotsessile vajalikke andmeid kliendi vahemälus ei ole. Samas on praktilistes hajusates failisüsteemides andme-, nime- ja staatusvahemälude tabamustegurid küllalt kõrged (vt. lk. 66). See tähendab, et klientprotsesside tegevus jääb serverprotsessile suures osas nähtamatuks ning serverprotsessi pilgu läbi võivad failikasutusmallid olla hoopis teistsugused kui nad tegelikult on. Intuitiivselt on lihtne taibata, et serverprotsessi jaoks ei pruugi failide, kataloogide ja failiatribuutide kasutamine enam ajaliselt lokaalne olla, sest kliendi vahemällu paigutatud andmetele viidatakse lähemas tulevikus väga tõenäoliselt uuesti. Kui klientprotsess viitab ühtedele ja samadele andmetele lühikese aja jooksul mitu korda, on serverprotsessile tihti nähtav vaid esimene viitamine, mille tagajärjel andmed tegelikult kliendi vahemällu paigutatakse. Järgmiste viitamiste käigus enam serverprotsessi poole ei pöörduta, sest andmed on juba kliendi vahemälust kättesaadavad. Kui failide, kataloogide ja failiatribuutide kasutamine pole serverprotsessi jaoks ajaliselt lokaalne, pole seda ka serverarvuti kettablokkide kasutamine.

Willick, Eager ja Bunt on viinud läbi hulga simulatsioone, mis selle mõttekäigu õigsust kinnitavad [Will92]. Simulatsioonide läbiviimisel tegid nad lihtsustava eelduse, et kliendi vahemälu koosneb vaid andmevahemälust ning serveri vahemälus hoitakse ainult failidesse kuuluvaid kettablokke. Lisaks eeldasid nad, et serverarvuti kettablokid on 8Kb suurused, kliendi vahemälu elementideks on 8Kb suurused failiblokid ning kliendi vahemälu asendamisstrateegiaks on LRU. Lähteandmed koguti UNIX-operatsioonisüsteemiga tööjaamadest ajal, mil neis töötasid lokaalseid faile intensiivselt kasutavad protsessid. Willick, Eager ja Bunt eeldasid veel, et hajus failisüsteem koosneb 1 klient- ja 1 serverarvutist, kuna eelnev analüüs ei näidanud, et tulemused sõltuksid simuleeritud klientarvutite hulgast. 1995. aastal läbiviidud uued

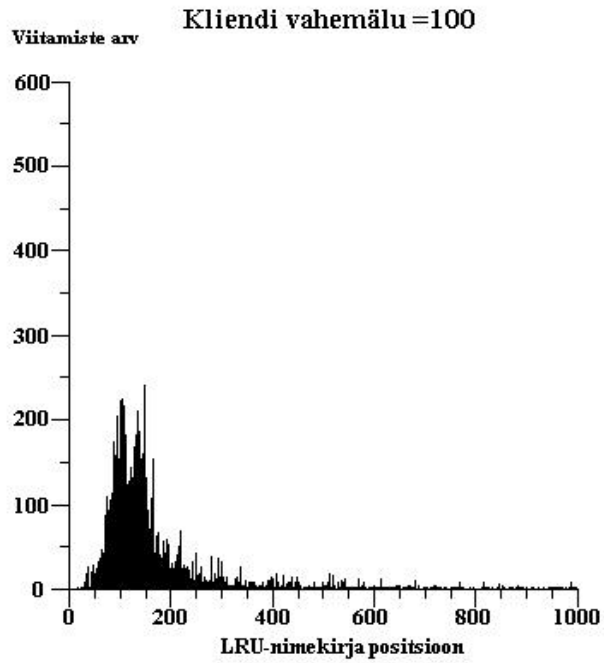
simulatsioonid, kus jäljendati 48 klient- ja 1 serverarvutist koosneva hajusa failisüsteemi tööd, kinnitasid Willicki, Eageri ja Bunti saadud tulemusi [Froese95].

Willick, Eager ja Bunt viisid simulatsioone läbi mitmesuguste kliendi vahemälu suuruste jaoks vahemikust 0-200 blokki, eeldades, et serveri vahemälu on lõpmatu suurusega. Serveri vahemälus asuvate blokkide kohta peeti LRU-nimekirja ja registreeriti, mitu korda igale LRU-nimekirja positsioonile (s.t. blokile, mis vastab antud positsioonile) simulatsiooni läbiviimise ajal viidati. Kui enamus viitamisi langeb LRU-nimekirja esimeste positsioonide arvele, näitab see kettablokkide kasutamise ajalist lokaalsust - enamasti viidatakse neile blokkidele, millele on lähiminevikus just viidatud. Kui LRU-nimekirja esimeste positsioonide arvele jääb vähe viitamisi, tõendab see, et kettablokkide kasutamine ei ole ajaliselt lokaalne.

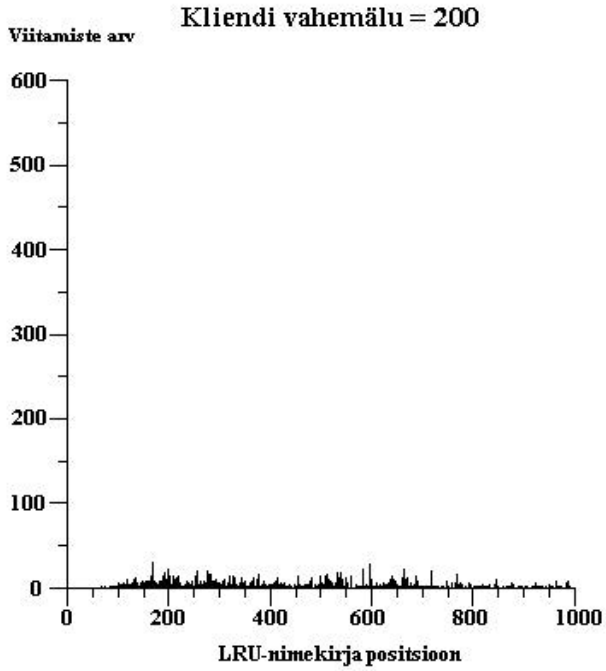
Willicki, Eageri ja Bunti saadud tulemused näitavad, et mida suurem on kliendi vahemälu, seda vähem ajaliselt lokaalne kettablokkide kasutamine on (vt. Joonis 3.2, Joonis 3.3 ja Joonis 3.4).



Joonis 3.2 Kettablokkide kasutamise ajaline lokaalsus, kui kliendi vahemälu puudub.



Joonis 3.3 Kettablokkide kasutamise ajaline lokaalsus, kui kliendi vahemälu on 100 bloki suurune.



Joonis 3.4 Kettablokkide kasutamise ajaline lokaalsus, kui kliendi vahemälu on 200 bloki suurune.

Kui kliendi vahemälu puudub, langeb LRU-nimekirja esimese positsiooni arvele üle 30000 viitamise (Joonis 3.2 seda ei kajasta, sest y-teljel asuv maksimaalne väärtus on 600). Kui kliendi vahemälu on 100 või 200 bloki suurune (vt. Joonis 3.3 ja Joonis 3.4), langeb LRU-nimekirja esimese viiekümne positsiooni arvele väga vähe viitamisi. See tähendab, et paigutamisstrateegia, kus andmeid paigutatakse vahemällu neile viitamisel, ja LRU-asendamisstrateegia efektiivsed vaid siis, kui kliendi vahemälu puudub (või on väga väike). Kuigi simulatsioonid viidi läbi vaid andmevahemälu jaoks, võib arvata, et need tulemused kehtivad ka nime- ja staatusvahemälude jaoks. Mida suuremad nad on, seda rohkem klientprotsessidele vajalikke andmeid nad mahutavad ning seda väiksem on tõenäosus, et serverprotsessi poole pöördutakse lühikese aja jooksul mitu korda ühtede ja samade andmete omandamiseks.

Willick, Eager ja Bunt püüdsid leida serveri vahemälu asendamisstrateegiat, mis töötaks hästi kliendi vahemälu suurusest sõltumata [Will92]. Nad vaatlesid otsingute käigus lähemalt LRU-, LFU-, FBR-, MIN- ja RAND-asendamisstrateegiaid, eeldades lihtsustavalt, et kliendi vahemälu koosneb vaid andmevahemälust, mille elementideks on blokid.

LFU-asendamisstrateegia (*Least Frequently Used replacement policy*) korral asendatakse blokk, mille kasutamise *sagedus* on kõige väiksem (s.t. millele on serveri vahemälus viibimise aja jooksul kõige vähem arv kordi viidatud). Koos iga blokiga hoitakse vahemälus ka bloki *viitamiste arvu* (*reference count*). Kui blokk paigutatakse serveri vahemällu, loetakse bloki viitamiste arv võrdseks ühega. Kui juba vahemälus asuvale blokile viidatakse, suurendatakse bloki viitamiste arvu ühe võrra. Kui uuele blokile ruumi tegemiseks on vaja mingi blokk vahemälust välja jätta, asendatakse selline blokk, mille viitamiste arv on kõige väiksem. Kui leidub mitu blokki, mille viitamiste arv on vähim, siis rakendatakse asendamisel *LRU-printsiipi* - selliste blokkide hulgast valitakse asendamiseks viimati kõige kauem aega tagasi kasutatud blokk. Kui kliendi vahemälu on väga väike (või puudub) ning kettablokkide kasutamine on ajaliselt lokaalne, on LFU-asendamisstrateegial serveri vahemälu asendamisstrateegiana kaks järgmist puudust [Will92]:

- **Mõne bloki viitamiste arv võib kasvada nii suureks, et seda blokki ei asendata kunagi.** Tänu blokkide kasutamise ajalisele lokaalsusele viidatakse blokkidele

lühikese aja jooksul palju kordi. Kui blokile on palju kordi viidatud lähemas minevikus, siis blokkide kasutamise ajalise lokaalsuse tõttu on küllalt tõenäoline, et talle viidatakse lähemas tulevikus uuesti ning bloki vahemällu jätmine on õigustatud. Samas võib bloki suur viitamiste arv pärineda kaugemast minevikust, mistõttu pole enam kindel, et blokile lähemas tulevikus uuesti viidatakse. Aja jooksul võib vahemällu koguneda rohkesti suure viitamiste arvuga blokke, millele lähemas tulevikus tõenäoliselt ei viidata ning mida pole kunagi võimalik teiste blokkidega asendada. Selle probleemi lahendamiseks modifitseerisid Willick, Eager ja Bunt LFU-asendamisstrateegiat nii, et bloki viitamiste arv loetakse teatud juhtudel *aegunuks*. Selleks hoitakse meeles viitamiste arvu aritmeetilist keskmist, mis leitakse üle kõigi vahemälus asuvate blokkide. Kui see keskmine ületab mingil hetkel teatud väärtuse A_{max}, siis iga bloki korral loetakse tema uueks viitamiste arvuks endise väärtuse C asemel väärtus $\lceil C/2 \rceil$ ($\lceil\ \rceil$ on arvule lähima temast suurema täisarvu leidmise operaator).

- **Mõnede blokkide viitamiste arv võib kasvada liiga suureks ja blokid jäävad seetõttu vahemällu liiga kauaks.** Tänu blokkide kasutamise ajalisele lokaalsusele viidatakse blokkidele lühikese aja jooksul palju kordi. LFU-asendamisstrateegia seisukohalt põhjustavad probleeme sellised blokid, mida kasutatakse lühikese aja jooksul väga intensiivselt ning seejärel pikema aja vältel üldse ei kasutata. Pärast lühikest ja intensiivset kasutusperioodi on selliste blokkide viitamiste arv nii suur, et nad jäetakse vahemälust välja hiljem, kui oleks otstarbekas. Willick, Eager ja Bunt leidsid, et bloki viitamiste arvu liigset suurust põhjustavad enamasti selle bloki osalised modifitseerimised. Kui klientprotsess modifitseerib andmevahemälust puuduvat blokki osaliselt, pöördub kliendimoodul bloki omandamiseks serverprotsessi poole, mille tagajärjel bloki viitamiste arv ühe võrra suureneb. Pärast bloki modifitseerimist saadab kliendimoodul selle bloki vastavalt kirjutamisstrateegiale teatud aja pärast serverprotsessile, mille tagajärjel bloki viitamiste arv veelkord ühe võrra suureneb. Seega suurendab üksainus bloki modifitseerimine bloki viitamiste arvu tegelikult kahe võrra. Willick, Eager ja Bunt modifitseerisid selle probleemi lahendamiseks LFU-asendamisstrateegiat nii, et bloki viitamiste arvu suurendataks ainult bloki lugemisel.

FBR-asendamisstrateegia (*Frequency Based Replacement policy*) püüab endas ühendada LRU- ja LFU-asendamisstrateegiate tugevaid külgi ning vältida nende nõrku külgi. FBR-asendamisstrateegia korral peetakse vahemälus asuvate blokkide kohta küll LRU-nimekirja, kuid blokke asendatakse nende kasutamise sageduste põhjal. Serveri vahemälu on jagatud *uueks*, *keskmiseks* ja *vanaks partitsiooniks* (*new*, *middle*, *old partition*). Nende partitsioonide suurused on määratud FBR-asendamisstrateegia kahe parameetriga F_{new} ja F_{old}. F_{new} näitab, kui suur osa vahemälust (protsentides) moodustab uue partitsiooni, ning F_{old} näitab, kui suur osa vahemälust (protsentides) moodustab vana partitsiooni. Uude partitsiooni kuulub F_{new}% blokkidest, mille nimed asuvad LRU-nimekirja alguses, ning vanasse partitsiooni F_{old}% blokkidest, mille nimed asuvad LRU-nimekirja lõpus. Uus partitsioon sisaldab seega kõige lähemas minevikus kasutatud blokke ning vana partitsioon kõige kaugemas minevikus kasutatud blokke. Keskmine partitsioon sisaldab kõiki ülejäänud blokke.

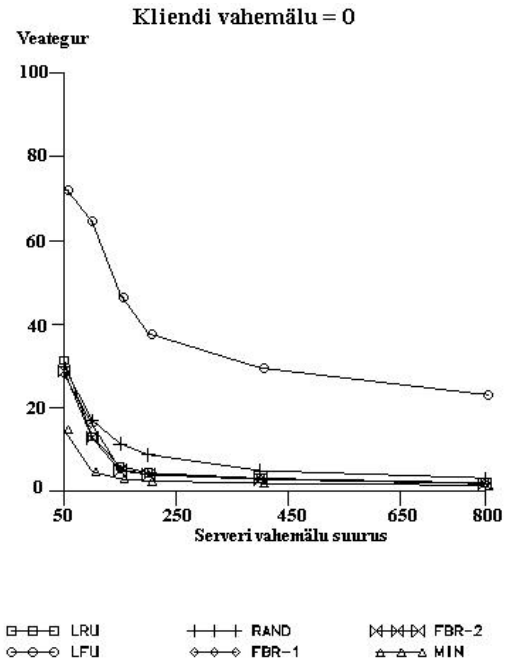
Kui viidatakse uude partitsiooni kuuluvale blokile, siis LFU-asendamisstrateegia puuduste kõrvaldamiseks bloki viitamiste arvu ei suurendata. Kuna uude partitsiooni kuuluvad hiljuti kasutatud blokid, siis välditakse sellega olukorda, kus bloki viitamiste arv kasvab blokkide kasutamise ajalise lokaalsuse tõttu liiga suureks. Ülejäänud partitsioonidesse kuuluvate blokkide korral suurendatakse blokile viitamisel selle viitamiste arvu ühe võrra. Kui vahemälus on vaja uue bloki jaoks ruumi teha, siis asendatakse vana partitsiooni vähima viitamiste arvuga blokk. Kui vanas partitsioonis leidub mitu vähima viitamiste arvuga blokki, siis nende hulgast asendatava bloki valimiseks rakendatakse LRU-printsiipi. Kuna asendatakse ainult vanasse partitsiooni kuuluvaid blokke, välditakse sellega olukorda, kus vahemälust jäetaks välja sinna lühikest aega tagasi paigutatud blokk, kuna ta viitamiste arv pole veel piisavalt suur.

Lisaks kasutab FBR-asendamisstrateegia veel LFU-asendamisstrateegia kaht parandust, kuna need suurendavad mõõtmiste kohaselt LFU-asendamisstrateegia efektiivsust [Will92]. On lihtne näha, et võttes $F_{new} = 0\%$ ja $F_{old} = 100\%$, osutub FBR-asendamisstrateegia LFU-asendamisstrateegiaks.

Willick, Eager ja Bunt vaatlesid veel MIN- ja RAND-asendamisstrateegiaid. Kui asendamisstrateegiate efektiivsust hinnatakse serveri vahemälu vea- ja tabamusteguri järgi, on MIN-asendamisstrateegia teoreetiline optimum (vt. lk. 66). RAND-asendamisstrateegia kohaselt asendatakse juhuslik blokk (kõigi vahemälu

blokkide asendamise tõenäosused on võrdsed). Kuna teised strateegiad püüavad oma otsuseid vastu võtta tunduvalt "intelligentsemate" kriteeriumide alusel, kui seda on juhuslikkus, saab RAND-asendamisstrateegiat kasutada teatud mõttes "alumise piirina" teiste asendamisstrateegiate efektiivsuse hindamisel.

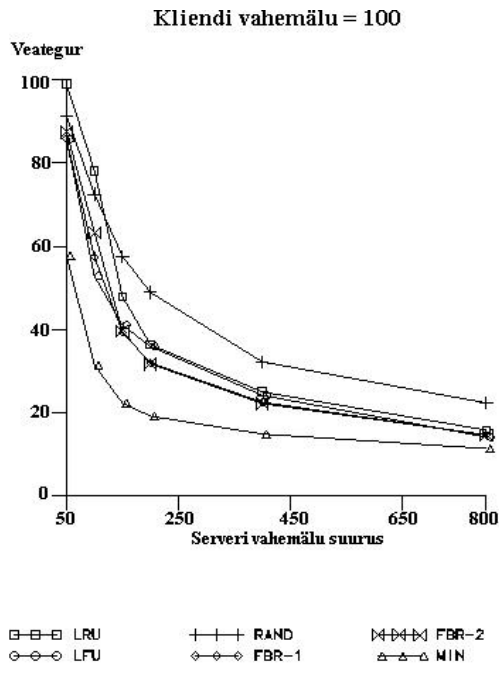
Nimetatud viie asendamisstrateegia efektiivsuse hindamiseks viisid Willick, Eager ja Bunt läbi hulga simulatsioone, mille eeldused on analoogilised lk. 74 toodud eeldustega. Simulatsioone viidi läbi 0-200 bloki suuruste kliendi vahemälude ja 50-800 bloki suuruste serveri vahemälude jaoks. Lähemalt vaadeldi FBR-asendamisstrateegia kaht varianti. FBR-1 korral võeti $F_{new} = 25\%$ ja $F_{old} = 60\%$, FBR-2 korral võeti $F_{new} = 25\%$ ja $F_{old} = 40\%$. Mõlema asendamisstrateegia korral võeti veel $A_{max} = 100$. Asendamisstrateegiate efektiivsust hinnati serveri vahemälu veateguri põhjal - mida madalamaks veategur osutus, seda efektiivsemaks asendamisstrateegiat loeti.



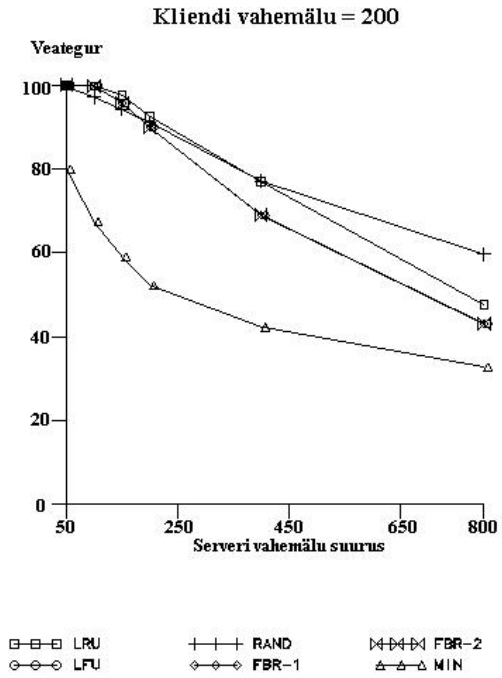
Joonis 3.5 Asendamisstrateegiate veategurid erineva suurusega serveri vahemälude korral, kui kliendi vahemälu puudub.

Kui kliendi vahemälu puudub (vt. Joonis 3.5), on LFU hoolimata kahest tehtud parandusest serveri vahemälu asendamisstrateegiaks äärmiselt ebasobiv - isegi RAND-asendamisstrateegia korral on serveri vahemälu veategur palju madalam. LRU-

asendamisstrateegia on seevastu väga efektiivne ning suurema serveri vahemälu korral optimumile väga lähedal. Need tulemused on loomulikud, sest kliendi vahemälu puudumisel on klientprotsesside tegevus serverprotsessile täielikult nähtav ning seetõttu on kettablokkide kasutamine ajaliselt lokaalne. FBR-asendamisstrateegiad on peaaegu sama efektiivsed kui LRU-asendamisstrateegia. Kui kliendi vahemälu suureneb (vt. Joonis 3.6 ja Joonis 3.7), siis peale LFU-asendamisstrateegia väheneb kõigi ülejäänud asendamisstrateegiate efektiivsus märgatavalt, eriti aga LRU oma. Kliendi vahemälu suurenemisel läheneb LFU oma efektiivsuselt FBR-asendamisstrateegiatele, olles suurima simuleeritud kliendi vahemälu korral nendega praktiliselt samaväärne. Kuna LRU-asendamisstrateegia efektiivsus langeb kliendi vahemälu suurenemisel järsult, leiab veelkord kinnitust hüpotees, et mida suurem on kliendi vahemälu, seda vähem ajaliselt lokaalne kettablokkide kasutamine on.



Joonis 3.6 Asendamisstrateegiate veategurid erineva suurusega serveri vahemälude korral, kui kliendi vahemälu on 100 bloki suurune.



Joonis 3.7 Asendamisstrateegiate veategurid erineva suurusega serveri vahemälude korral, kui kliendi vahemälu on 200 bloki suurune.

Simulatsioonide tulemused näitavad, et LFU-asendamisstrateegia, kus asendatav blokk valitakse blokkide kasutamise sageduste põhjal, on suurema kliendi vahemälu puhul efektiivsem kui LRU-asendamisstrateegia. See tähendab, et suurema kliendi vahemälu puhul viidatakse suurema tõenäosusega neile serveri vahemälus asuvatele blokkidele, mida on kasutatud *sagedamini* (s.t. vahemälus viibimise aja jooksul rohkem arv kordi), mitte aga neile blokkidele, mida on kasutatud *lähiminevikus*. Samas on hajusas failisüsteemis võrdlemisi suured kliendi vahemälud üsna tüüpilised - tavaliselt eraldatakse operatiivmälus paikneva kliendi vahemälu jaoks mitte vähem kui 10% klientarvuti operatiivmälust [Baker91] ning klientarvuti kettal paiknev kliendi vahemälu on enamasti vähemalt 20Mb suurune [Muntz92]. Seetõttu võib väita, et LFU on serveri vahemälu asendamisstrateegiaks paremini sobiv kui LRU-asendamisstrateegia. Parimad serveri vahemälu asendamisstrateegiad on hübriidsed FBR-asendamisstrateegiad, mis on võrdlemisi efektiivsed kõigi kliendi vahemälu suuruste korral.

Samas on näha (vt. Joonis 3.7), et suurema kliendi vahemälu korral pole ükski siin vaadeldud asendamisstrateegia teoreetilisele optimumile kuigi lähedal. Seetõttu

väärivad serveri vahemälu asendamisstrateegiad edasist uurimist ning suuremat tähelepanu, kui neile seni on osaks langenud [Froese95].

3.3 Kirjutamisstrateegiad

Kliendi vahemälu osade (andme-, nime- ja staatusvahemälu) kirjutamisstrateegiad määravad, kui kaua klientprotsessi poolt failidesse, kataloogidesse ja failiatribuutidesse tehtud muudatusi enne serverprotsessile saatmist kliendi vahemälus hoitakse. Tehtud muudatusi edastab serverprotsessile kliendimoodul, kasutades kaugprotseduuridena realiseeritud serverprotsessi operatsioone (vt. AFSi kohta toodud näidet, Tabel 3.4). Serveri vahemälu kirjutamisstrateegia määrab, kui kaua serverprotsessi poolt modifitseeritud kettablokke enne serverarvuti kettale kirjutamist serveri vahemälus hoitakse.

Failid	`Store`
Kataloogid	`Remove, Create, Rename, Symlink, Link, Makedir, Removedir`
Failiatribuudid	`Store, (Remove, Create, Rename, Symlink, Link, Makedir, Removedir)`

Tabel 3.4 AFSi serverprotsessi operatsioonid, mille abil kliendimoodul failidesse, kataloogidesse ja failiatribuutidesse tehtud muudatusi serverprotsessile edastab (sulgudes toodud operatsioonid muudavad failiatribuute kõrvalefektina).

Nimevahemälu puhul kasutavad peaaegu kõik praktilised hajusad failisüsteemid kirjutamisstrateegiana *läbikirjutamist* (*write-through*) - kui klientprotsess modifitseerib mõnd kataloogifaili (süsteemifunktsioonide `creat`, `link`, `rmdir` jms. abil), saadetakse muudatused serverprotsessile viivitamatult. Läbikirjutamine on levinuim nimevahemälu kirjutamisstrateegia, sest selle kasutamisel saavad kataloogipuus toimunud muutused serverprotsessile kohe nähtavaks ning väheneb tõenäosus, et failinimede ruum on erinevates hajusa failisüsteemi sõlmedes töötavate klientprotsesside pilgu läbi erinev [Sat93]. Teatud juhtudel on võimatu tehtud muudatusi mõnda aega ainult nimevahemälus hoida, ilma et koheselt serverprotsessi poole pöördutaks. Kui klientprotsess modifitseerib kataloogifaili, luues uue faili, kataloogi, viite või sümbolviite, siis kataloogifaili uue kirje nimevahemällu salvestamiseks on vaja teada

loodud objekti identifikaatorit. Selle omandamiseks peab kliendimoodul aga kõigepealt objekti loomise operatsiooni abil serverprotsessi poole pöörduma.

Staatusvahemälu puhul kasutatakse kirjutamisstrateegiana läbikirjutamise modifikatsiooni. Kui klientprotsess muudab faili atribuute otseselt (süsteemifunktsioonide `chown`, `chmod` jms. abil), saadetakse muudatused kohe ka serverprotsessile. Nii tehakse paljuski sellepärast, et muutused faili juurdepääsunimekirjas serverprotsessile viivitamatult nähtavaks saaksid ning oleks välistatud faili kasutamine volitamata kasutaja poolt.

Failide atribuudid võivad kõrvalefektina muutuda ka siis, kui klientprotsess katalooge modifitseerib. Näiteks failile F osutava viite loomine kataloogis D (süsteemifunktsiooni `link` abil) muudab kataloogi D viimase kasutamise ja modifitseerimise aegu, kataloogifaili suurust ning kataloogi atribuutide viimase modifitseerimise aega; samuti muudab see faili F viidete arvu ning faili atribuutide viimase modifitseerimise aega. Kliendimoodul ei edasta sedalaadi muudatusi serverprotsessile eraldi operatsiooni abil. Kui kliendimoodul pöördub mingi kataloogi modifitseeriva operatsiooni abil serverprotsessi poole, muudab serverprotsess ise ka vajalikke failiatribuute. Kuna nimevahemälu kirjutamisstrateegiana kasutatakse läbikirjutamist ning kataloogipuus toimunud muudatused saadetakse serverprotsessile viivitamatult, siis tähendab see, et ka kataloogide modifitseerimisel atribuutidega toimunud muudatused saadetakse serverprotsessile viivitamatult.

Mõnedel juhtudel pole siiski otstarbekas atribuutidega toimunud muudatusi viivitamatult serverprotsessile edastada. Kui klientprotsess loeb failist süsteemifunktsiooni `read` abil või kataloogifailist süsteemifunktsiooni `readdir` abil, muudab see faili või kataloogifaili viimase kasutamise aega. Kui klientprotsess süsteemifunktsiooni `write` abil faili kirjutab, muudab see faili viimase kasutamise ja viimase modifitseerimise aegu ning atribuutide viimase modifitseerimise aega. Kui faili kirjutamisel faili suurus muutub, muudab kirjutamine veel faili suurust näitavat atribuuti. Kui kõigil neil juhtudel samuti läbikirjutamist rakendataks, tooks iga failist või kataloogifailist lugemine ja faili kirjutamine kaasa pöördumise serverprotsessi poole. Seetõttu kasutatakse näiteks NFSi puhul lahendust, kus serverprotsess muudab failiga seotud ajalisi atribuute ja faili suurust näitavat atribuuti siis, kui kliendimoodul operatsioonide `read`, `write` või `readdir` abil serverprotsessi poole pöördub. AFSi

puhul, kus faili saab ainult koos atribuutidega kliendi vahemällu paigutada, kasutatakse lahendust, kus faili kasutamise käigus muutunud atribuudid saadetakse serverprotsessile faili sulgemisel (vt. Tabel 2.3, operatsioon `Store`).

Erinevalt nime- ja staatusvahemälust pole andmevahemälu puhul läbikirjutamine kuigi otstarbekas kirjutamisstrateegia, sest faili kirjutatud uute baitide eluiga on lühike (vt. alapunkt 3.1.1). See tähendab, et paljusid failidesse tehtud muudatusi pole mõtet serverprotsessile saata, sest lähemas tulevikus kirjutatakse need nagunii üle või kustutatakse. Sellepärast on andmevahemälu kirjutamisstrateegiana mõttekam kasutada *viivitatud kirjutamist* (*delayed-write*) - failidesse tehtud uuendused salvestatakse andmevahemällu, kus neid enne serverprotsessile saatmist mõnda aega hoitakse. Viivitatud kirjutamise puhul on serverarvuti ja sellega ühendatud kohtvõrgu koormus väiksem - faili andmevahemälus asuvaid baite võidakse mitu korda modifitseerida, ilma et toimuks ühtegi pöördumist serverprotsessi poole. Viivitatud kirjutamise puuduseks on see, et ta pole *usaldatav* (*reliable*) - klientarvuti töö avariilõpu puhul võib andmeid kaduma minna, sest erinevalt läbikirjutamisest tekib olukord, kus failidesse tehtud uuendused asuvad mõnda aega *ainult* andmevahemälus.

Kõigis praktilistes hajusates failisüsteemides kasutatakse lähenemist, kus andmevahemällu salvestatud muudatuste serverprotsessile edastamiseks saadetakse serverprotsessile vastavad modifitseeritud andmevahemälu elemendid. Serverprotsessile saadetakse terve element ka siis, kui elementi on osaliselt modifitseeritud [Coul94, Howard88, Nelson87].

Kui andmevahemälu elementideks on failid, kasutatakse tavaliselt kirjutamisstrateegiat, kus modifitseeritud fail saadetakse serverprotsessile faili sulgemisel. Sellist andmevahemälu kirjutamisstrateegiat kasutavad näiteks AFS [Howard88] ja Coda [Sat93].

Kui andmevahemälu elementideks on failiblokid, on võimalused kirjutamisstrateegia valikuks tunduvalt suuremad. Failiblokke sisaldava andmevahemälu ja serveri vahemälu kirjutamisstrateegiaid on seni kõige põhjalikumalt uurinud Nelson [Nelson88]. Andmevahemälu kirjutamisstrateegiat nimetas ta *kliendi kirjutamisstrateegiaks* (*client writing policy*) ning serveri vahemälu kirjutamisstrateegiat *serveri kirjutamisstrateegiaks* (*server writing policy*). Neid termineid kasutatakse Nelsoni saadud tulemuste kirjeldamisel edaspidi ka selles töös.

Olulisemad kliendi kirjutamisstrateegiad on järgmised [Nelson88]:

- **WT** (*write-through*) ehk läbikirjutamine - faili kirjutamise operatsioonist (süsteemifunktsioonist `write`) pöördutakse tagasi siis, kui operatsiooni käigus modifitseeritud blokid on serveri vahemällu kirjutatud.
- **WBOC** (*write-back-on-close*) - modifitseeritud blokid saadetakse serverprotsessile faili sulgemisel. Faili kirjutamise operatsioonist pöördutakse tagasi kohe, kui muudatused on andmevahemällu kirjutatud. Faili sulgemise operatsioonist (süsteemifunktsioonist `close`) ei pöörduta seevastu tagasi enne, kui faili kuuluvad modifitseeritud blokid on serveri vahemällu kirjutatud.
- **ASAP** (*as-soon-as-possible*) - faili kirjutamise operatsioonist pöördutakse tagasi kohe, kui muudatused on andmevahemällu kirjutatud. Modifitseeritud blokk kirjutatakse asünkroonselt serveri vahemällu niipea, kui on uuendatud selle bloki kõiki baite. Kui suletakse uuendatud fail, kirjutatakse serveri vahemällu kõik faili kuuluvad modifitseeritud blokid, mida pole veel mingil põhjusel sinna kirjutatud. Faili sulgemise operatsioonist pöördutakse viivitamatult tagasi, ootamata modifitseeritud blokkide serveri vahemällu kirjutamist.
- **WBOC-ASAP** - WBOC- ja ASAP-strateegia kombinatsioon. Erineb ASAP-strateegiast vaid selle poolest, et faili sulgemise operatsioonist ei pöörduta tagasi enne, kui serverprotsessile saadetud modifitseeritud blokid on serveri vahemällu kirjutatud. Seda kliendi kirjutamisstrateegiat kasutavad NFSi versioonid, kus on realiseeritud bio-deemonid (vt. lk. 40).
- **D-30** (*delay-30*) - faili kirjutamise operatsioonist pöördutakse tagasi kohe, kui muudatused on andmevahemällu kirjutatud. Andmevahemälu vaadatakse iga 5 sekundi tagant läbi ning serveri vahemällu kirjutatakse asünkroonselt need modifitseeritud blokid, mida pole viimase 30 sekundi jooksul enam muudetud. Seda kliendi kirjutamisstrateegiat kasutab Sprite [Nelson88].

Kõigi kliendi kirjutamisstrateegiate puhul saadetakse modifitseeritud blokk serverprotsessile ka siis, kui see blokk andmevahemälust välja jäetakse, sest muidu läheks selle bloki sisu kaduma.

Olulisemad serveri kirjutamisstrateegiad on järgmised [Nelson88]:

- **D-30** - serveri vahemälu vaadatakse iga 5 sekundi tagant läbi ning serverarvuti kettale kirjutatakse need blokid, mida pole viimase 30 sekundi jooksul enam uuendatud. Seda serveri kirjutamisstrateegiat kasutab Sprite [Nelson88].
- **WT** - kui kliendimoodul pöördub modifitseeritud andmete serverprotsessile saatmiseks kaugprotseduuri poole ning serverprotsess kaugprotseduuri täitmise käigus serveri vahemälus asuvaid kettablokke modifitseerib, kirjutatakse uuendatud kettablokid viivitamatult serverarvuti kettale. Kaugprotseduuri täitmine lõpeb siis, kui blokkide kettale kirjutamine on lõppenud. Seda serveri kirjutamisstrateegiat kasutab NFS [Coul94].

Iga serveri kirjutamisstrateegia puhul kirjutatakse serveri vahemälus asuv modifitseeritud kettablokk serverarvuti kettale alati ka siis, kui see kettablokk vahemälust välja jäetakse.

Et kindlaks teha, kui efektiivsed mainitud kliendi ja serveri kirjutamisstrateegiad hajusa failisüsteemi skaleeritavuse ja klientprotsesside töökiiruse suurendamisel on, testis Nelson hajusa failisüsteemi Sprite prototüüpi. Testimiseks loodud prototüübi installatsioon koosnes 1 klient- ja 1 serverarvutist, mis olid omavahel kohtvõrgu abil ühendatud. Serveri vahemälu elementideks olid 4Kb suurused kettablokid ning kliendi vahemälu[12] elementideks 4Kb suurused failiblokid. Mingi kliendi ja serveri kirjutamisstrateegiate paari efektiivsuse hindamiseks modifitseeris Nelson vastavalt prototüüpi (normaalselt kasutab Sprite nii kliendi kui serveri kirjutamisstrateegiana D-30-kirjutamisstrateegiat). Nii kliendi kui serveri vahemälu olid 8Mb suurused (normaalselt on hajusas failisüsteemis Sprite vahemälude suurused muutuvad, vt. alapunkt 3.5.1). Vahemälud olid nii suured, et nad ei saanud kunagi täis ning ühegi testi täitmisel ei tulnud kunagi asendamisstrateegiat rakendada. See võimaldas vältida olukordi, kus kirjutamisstrateegiat ollakse sunnitud rakendama vahemälust väljajäetavate blokkide salvestamiseks. Iga testi täitmise lõppedes saadeti serverprotsessile need kliendi vahemälu modifitseeritud blokid, mida polnud veel jõutud serverprotsessile saata, ja serverarvuti kettale kirjutati need serveri vahemälus asuvad modifitseeritud kettablokid, mida polnud veel jõutud sinna kirjutada. Kuna reaalses elus oleks need vahemäludes sisalduvad modifitseeritud andmed mingil hetkel

[12] Nelson kasutab oma töös [Nelson88] termini *andmevahemälu* asemel terminit *kliendi vahemälu*, sest hajusas failisüsteemis Sprite koosneb kliendi vahemälu ainult andmevahemälust (vt. lk. 56). Töö selles osas (punkt 3.3) on edaspidi järgitud Nelsoni eeskuju.

serverprotsessile saadetud või serverarvuti kettale kirjutatud, annab see serverarvuti ja võrgu koormusest tõele lähedasema pildi.

Testid, mille alusel Nelson kirjutamisstrateegiate efektiivsust hindas (vt. Tabel 3.5), kasutavad kaugeid faile võrdlemisi intensiivselt. Et hinnata, kui efektiivsed on erinevad kirjutamisstrateegiad hajusa failisüsteemi skaleeritavuse suurendamisel, mõõtis Nelson testimise ajal võrgu ning serverarvuti protsessori ja ketta koormust. Hindamaks, kui efektiivsed on erinevad kirjutamisstrateegiad klientprotsesside töökiiruse suurendamisel, mõõtis ta testide täitmise aega.

Andrew test - kopeerida 70 programmifaili ja 200Kb andmeid sisaldav kataloogipuu ühest kataloogist teise; lugeda kataloogipuu kõigi failide atribuute; lugeda kataloogipuu kõigi failide kõiki baite; lõpuks kompileerida ja linkida kõik failid.
Vm-make-test - kompileerida C-keeles kirjutatud Sprite'i virtuaalmälusüsteem, mille kood sisaldub 15 failis ning on 11250 rea pikkune.
Sort-test - sorteerida 1Mb suurune fail (`sort`-utiliidi abil).

Tabel 3.5 Kirjutamisstrateegiate efektiivsuse hindamisel kasutatud testid.

3.3.1 Kirjutamisstrateegiate mõju võrgu koormusele

Võrgu koormus sõltub loomulikult ainult kliendi kirjutamisstrateegiast. Nelson hindas võrgu koormust klient- ja serverarvuti vahel mööda võrku edastatud andmete hulga järgi. Võrgu koormuse seisukohalt on halvim WT-kirjutamisstrateegia, sest selle kasutamisel toob iga kirjutamisoperatsioon kaasa pöördumise serverprotsessi poole, et edastada operatsiooni käigus modifitseeritud blokke. Kuna ASAP- ja WBOC-ASAP-kirjutamisstrateegiad erinevad vaid selle poolest, kas faili sulgemisel kirjutatakse faili kuuluvad modifitseeritud blokid serveri vahemällu asünkroonselt või mitte, on võrgu koormus nende kirjutamisstrateegiate korral ühesugune. Kuna faili kirjutatakse enamasti järjestikuselt (vt. alapunkt 3.1.1) ja failiakent seega tagasi ei nihutata, siis WBOC-kirjutamisstrateegia kasutamisel saadetakse faili sulgemisel serverprotsessile tavaliselt samapalju blokke kui oleks saadetud ASAP- või WBOC-ASAP-kirjutamisstrateegia puhul faili avamise hetkest alates. Nelsoni mõõtmistulemused kinnitavad seda [Nelson88].

WBOC-, ASAP- ja WBOC-ASAP-kirjutamisstrateegiad on WT-strateegiast efektiivsemad juhul, kui suur osa kirjutamisoperatsioonidest on sellised, mille käigus uuendatakse failiblokke osaliselt. Nelson leidis, et kasutatud kolmest testist on Sort-test ainus, mille täitmisel modifitseeritakse failiblokke tavaliselt tervikuna. Selle testi puhul osutus WT-kirjutamisstrateegia sama efektiivseks kui WBOC-, ASAP- ja WBOC-ASAP-strateegiad [Nelson88].

D-30-kirjutamisstrateegia on Nelsoni mõõtmistulemuste põhjal palju efektiivsem kui WBOC-, ASAP- ja WBOC-ASAP-strateegiad - selle kasutamisel vähenes võrgu koormus kõigi testide puhul üle 30% [Nelson88]. Põhjus on selles, et D-30-strateegia kasutamisel õnnestub vältida paljude selliste blokkide korduvat serverprotsessile saatmist, mis lühikese aja jooksul mitu korda üle kirjutatakse; blokke, mis pärast nende loomist lühikese aja pärast kustutatakse, ei saadeta serverprotsessile üldse. Nelson tegi kindlaks, et enamik testide täitmise käigus loodud ajutisi faile kustutati vähem kui 30 sekundit pärast nende loomist ning D-30-strateegia kasutamisel kuulus suurem osa serverprotsessile saadetud blokkidest testi täitmise lõpptulemusena loodud failidesse.

3.3.2 Kirjutamisstrateegiate mõju klientprotsesside töökiirusele

Klientprotsesside töökiiruse seisukohalt on halvim kliendi kirjutamisstrateegia WT-strateegia, sest selle kasutamisel toob iga kirjutamisoperatsioon kaasa pöördumise serverprotsessi poole. WBOC-kirjutamisstrateegia puhul ei pöörduta serverprotsessi poole küll igal kirjutamisel, kuid faili sulgemise operatsioon on aeganõudev, sest serverprotsessile saadetakse kõik faili kuuluvad modifitseeritud blokid. WBOC-ASAP-kirjutamisstrateegia on efektiivsem kui WBOC-strateegia, sest siin jõutakse tavaliselt osa vajalikest blokkidest enne faili sulgemist asünkroonselt serverprotsessile saata ning faili sulgemise operatsioon toimub seetõttu kiiremini. Klientprotsesside töökiiruse seisukohalt on kõige efektiivsemad täielikult asünkroonse iseloomuga ASAP- ja D-30-kirjutamisstrateegiad, sest nende kasutamisel pöördutakse faili kirjutamise ja faili sulgemise operatsioonidest tagasi kohe, kui muudatused on kliendi vahemällu kirjutatud.

Nelson leidis, et kui serveri kirjutamisstrateegiaks on D-30-strateegia, ei erine testi täitmise ajad erinevate kliendi kirjutamisstrateegiate puhul üksteisest kuigi palju (vt. Tabel 3.6). Põhjus on selles, et kliendimooduli poolt väljakutsutud kaugprotseduuridest pöördutakse tagasi, ootamata serverprotsessile saadetud blokkide serverarvuti kettale kirjutamist. Nelson leidis ka, et kui serveri kirjutamisstrateegiana kasutatakse WT-strateegiat, on testi täitmise aja ja kliendi kirjutamisstrateegia vaheline sõltuvus märgatavam (vt. Tabel 3.6). Kui kliendi kirjutamisstrateegiaks on WT-, WBOC-, või WBOC-ASAP-strateegia ning faili kirjutamine või selle sulgemine toob kaasa pöördumise serverprotsessi poole, peab testprogramm ootama, kuni saadetud blokkide serverarvuti kettale kirjutamine lõpule viiakse. Testi täitmise aeg on seetõttu pikem kui teiste kliendi kirjutamisstrateegiate puhul.

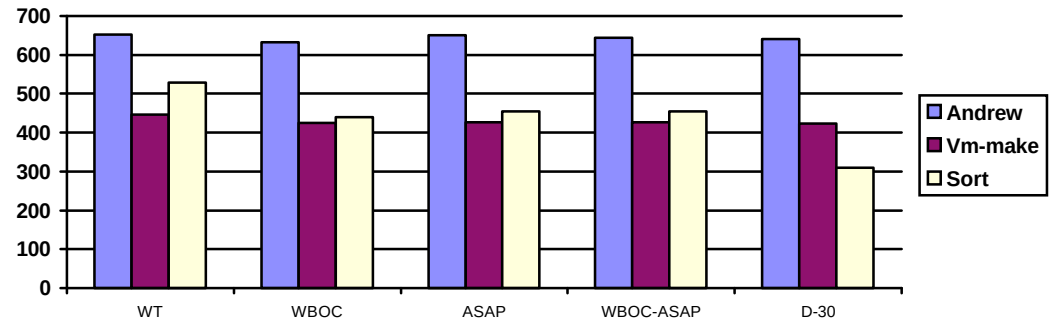
	Andrew test		**Vm-make-test**		**Sort-test**	
Serveri kirjutamis strateegia / **Kliendi kirjutamis strateegia**	**WT**	**D-30**	**WT**	**D-30**	**WT**	**D-30**
WT	424	279	370	300	124,5	66,1
WBOC	378	276	359	299	124,6	65,2
ASAP	316	273	313	297	65,5	62,5
WBOC-ASAP	345	273	330	297	87,4	62,5
D-30	295	269	308	291	70,0	58,7

Tabel 3.6 Testide täitmise ajad sekundites.

3.3.3 Kirjutamisstrateegiate mõju serverarvuti koormusele

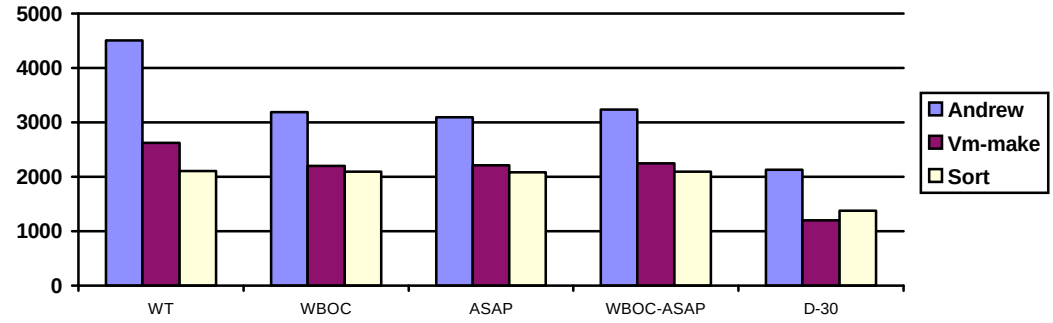
Serverarvuti ketta koormust hindas Nelson bloki kettale kirjutamise operatsioonide arvu järgi. Kui serveri kirjutamisstrateegiana kasutada D-30-strateegiat, on serverarvuti ketas võrdlemisi vähe koormatud. Kuigi kord juba modifitseeritud kettablokke kirjutatakse tihti uuesti üle (vt. alapunkt 3.1.1), toimub enamus ülekirjutamisi serveri vahemälus ilma ketta poole pöördumata. Nelson täheldas, et D-30-strateegia on serveri kirjutamisstrateegiana küllalt vähetundlik - kettale kirjutamiste arv jäi enam-vähem samaks kõigi kliendi kirjutamisstrateegiate korral (vt. Joonis 3.8). Erandiks oli Sort-test,

kus D-30 kliendi kirjutamisstrateegia kasutamisel toimus tunduvalt vähem kettale kirjutamisi kui teiste kliendi kirjutamisstrateegiate korral.



Joonis 3.8 Serverarvuti kettale kirjutamiste arv, kui serveri kirjutamisstrateegiaks on D-30.

Kui serveri kirjutamisstrateegiaks on WT-strateegia, on serverarvuti ketta koormus märgatavalt suurem, sest iga serverprotsessi poole pöördumine toob kaasa kettapöörduse. Isegi kui testimisel kasutati kliendi kirjutamisstrateegiana D-30-strateegiat, mille korral saadetakse serverprotsessile kõige vähem blokke, oli ketta koormus (võrreldes juhuga, kus serveri kirjutamisstrateegiaks on D-30-strateegia) kõigi testide puhul keskmiselt 3 korda suurem, ülejäänud kliendi kirjutamisstrateegiate korral kõigi testide puhul aga 5-7 korda suurem (vt. Joonis 3.9).



Joonis 3.9 Serverarvuti kettale kirjutamiste arv, kui serveri kirjutamisstrateegiaks on WT.

Nelsoni mõõtmistulemuste põhjal on WT-strateegia halvim kliendi kirjutamisstrateegia. Erandiks on Sort-testi korral saadud tulemused, sest `sort`-utiliit ei modifitseeri failiblokke peaaegu kunagi osaliselt. Seetõttu edastatakse WT-, WBOC-, ASAP- ja

WBOC-ASAP-kirjutamisstrateegiate korral serverprotsessile ühepalju blokke (vt. lk. 89) ja toimub ühepalju kettale kirjutamisi.

Kui serveri kirjutamisstrateegiana kasutada WT-strateegiat, peaks serverarvuti protsessor teoreetiliselt rohkem koormatud olema kui D-30-strateegia kasutamisel, sest WT-strateegia puhul on serverarvuti kettale kirjutamiste arv suurem ning sellele kulub rohkem protsessoriaega. Samas ei täheldanud Nelson Andrew ja Vm-make-testide puhul, et protsessori *utiliseeritus* (*utilization*)[13] serveri kirjutamisstrateegiast sõltuks [Nelson88]. Kuigi WT-strateegia korral toimus testide täitmisel rohkem kettale kirjutamisi ja kulus rohkem serverarvuti protsessoriaega, olid testide täitmise ajad pikemad (vt. Tabel 3.6) ja protsessori utiliseeritus jäi seetõttu enam-vähem samaks kui D-30-strateegia puhul. Kui kliendi kirjutamisstrateegiana kasutati WT-, WBOC-, ASAP- või WBOC-ASAP-strateegiat, siis Andrew testi täitmisel oli serverarvuti protsessori utiliseeritus mõlema serveri kirjutamisstrateegia korral 11-13%, Vm-make-testi täitmisel aga 9-10%. Kui kliendi kirjutamisstrateegiana kasutati D-30-strateegiat, mille korral serverprotsessile vähem blokke saadetakse, oli serverarvuti protsessor mõnevõrra vähem koormatud - Andrew testi täitmisel oli serverarvuti protsessori utiliseeritus mõlema serveri kirjutamisstrateegia puhul 11%, Vm-make-testi täitmisel 8%.

Ainult Sort-testi korral täheldas Nelson, et WT-strateegia kasutamisel serveri kirjutamisstrateegiana oli serverarvuti protsessor rohkem koormatud kui D-30-strateegia kasutamisel [Nelson88]. D-30-strateegia kasutamisel oli serverarvuti protsessori utiliseeritus kõigi kliendi kirjutamisstrateegiate puhul 7-11%, WT-strateegia kasutamisel aga 10-18%. Mõlemal juhul oli serverarvuti protsessor vähim koormatud, kui kliendi kirjutamisstrateegiaks oli D-30-strateegia.

3.3.4 Kokkuvõte

Halvim kliendi kirjutamisstrateegia on WT-strateegia. Selle kasutamisel on serverarvuti ketas ja serverarvutiga ühendatud kohtvõrk kõige rohkem koormatud ning

[13] Nelson pole oma töös [Nelson88] seda mõistet defineerinud. Utiliseerituse all mõistab ta ilmselt testi täitmisel kulunud serverarvuti protsessoriaja ja testi täitmise aja suhet. See näitab, kui suur on testimisel serverarvuti protsessori keskmine koormus ajaühikus.

klientprotsesside töökiirus on kõige väiksem. Efektiivseim kliendi kirjutamisstrateegia on D-30-strateegia.

WBOC-, ASAP- ja WBOC-ASAP-strateegiad on oma efektiivsuselt WT- ja D-30-strateegiate vahepealsed. Nelsoni mõõtmistulemuste kohaselt pole neil strateegiatel serverarvuti ja sellega ühendatud kohtvõrgu koormuse seisukohalt erilist vahet. ASAP- ja WBOC-ASAP-kirjutamisstrateegiad võimaldavad oma asünkroonse iseloomu tõttu klientprotsessidel kiiremini töötada; WBOC-strateegia kasutamisel on faili sulgemise operatsioon ajaliselt kulukam, mis vähendab klientprotsesside töökiirust.

WT-, WBOC- ja WBOC-ASAP-strateegiatel on see eelis, et klientprotsessidel on võimalik kontrollida, kas modifitseeritud blokid saadeti serverprotsessile või mitte. Kui modifitseeritud blokke ei õnnestu serveri vahemällu kirjutada, siis WT-strateegia puhul tagastab süsteemifunktsioon `write` veakoodi ning WBOC- ja WBOC-ASAP-strateegiate puhul tagastab süsteemifunktsioon `close` veakoodi. Kontrollides nende süsteemifunktsioonide poolt tagastatud väärtusi, saavad klientprotsessid blokkide edastamise käigus tekkinud vigu kindlaks teha ja neid töödelda.

D-30-strateegia on serveri kirjutamisstrateegiana tunduvalt efektiivsem kui WT-strateegia. Nelsoni mõõtmised on näidanud ka selle strateegia vähetundlikkust - ebaefektiivsete kliendi kirjutamisstrateegiate kasutamisel suureneb serverarvuti koormus ja väheneb klientprotsesside töökiirus mitte eriti suurel määral. Võib siiski arvata, et suuremas hajusas failisüsteemis ei suuda D-30-strateegia ebaefektiivsete kliendi kirjutamisstrateegiate negatiivset mõju enam kuigivõrd vähendada.

WT-kirjutamisstrateegia on serveri kirjutamisstrateegiana küll väheefektiivne, kuid tema eeliseks on usaldatavus - kliendimooduli poolt serverprotsessile saadetud andmed kirjutatakse kohe serverarvuti kettale, et vältida nende kadumaminekut serverarvuti töö avariilõpu korral. Seetõttu kasutatakse WT-strateegiat näiteks NFSi puhul koos WBOC-ASAP kliendi kirjutamisstrateegiaga. Kui süsteemifunktsioon `close` faili sulgemisel veakoodi ei tagasta, võib klientprotsess kindel olla, et tema poolt modifitseeritud faili blokid on serverarvuti kettale kirjutatud ja nende kadumaminek on välistatud.

3.4 Valideerimisstrateegiad

Hajusa failisüsteemi töö käigus juhtub tihti, et erinevate klientarvutite juures asuvates kliendi vahemäludes hoitakse ühtesid ja samu andmeid sisaldavaid elemente (näiteks ühe ja sama faili blokke). Kui ühes klientarvutis töötav klientprotsess mingis taolises elemendis sisalduvaid andmeid modifitseerib ja tehtud muudatused kirjutamisstrateegia kohaselt serverprotsessile saadetakse, muutub teistes kliendi vahemäludes asuv samu andmeid sisaldav element mittekooskõlaliseks ja tuleb kehtetuks tunnistada. Andme-, nime- ja staatusvahemälu valideerimisstrateegiad määravad, kuidas kliendimoodul kindlaks teeb, millised vahemälu elemendid on mittekooskõlaliseks muutunud.

Andme- ja nimevahemälu valideerimine toimub tavaliselt *failide alusel* - seda ka siis, kui vahemälu elementideks on blokid. Kui faili või kataloogifaili on modifitseeritud, loetakse mittekooskõlaliseks *kõik* selle faili baite sisaldavad andme- või nimevahemälude elemendid (välja arvatud selle klientarvuti juures asuva vahemälu elemendid, kus töötab faili modifitseerinud klientprotsess). Failide alusel toimuv andme- ja nimevahemälu valideerimine on praktilisem kui faili üksikute blokkide alusel toimuv, sest nii väheneb vahemälude valideerimiseks vajaliku info hulk, mida serverprotsess peab haldama. Serverprotsess ei pea meeles pidama infot iga üksiku bloki kohta (näiteks selle viimase modifitseerimise aeg), vaid ainult iga faili kohta. Blokkide alusel toimuval andmevahemälu valideerimisel pole nagunii erilist mõtet, sest faile modifitseeritakse enamasti tervikuna (vt. alapunkt 3.1.1).

Töö selle punkti alapunktides vaadeldakse levinumaid valideerimisstrateegiaid ja nende modifikatsioone.

3.4.1 Kliendi juhitud valideerimine

Et kindlaks teha, millised vahemälu elemendid on mittekooskõlaliseks muutunud, pöördub kliendimoodul *kliendi juhitud valideerimise* (*client driven validation*) korral ise serverprotsessi poole. Selle valideerimisstrateegia puhul kontrollib kliendimoodul ainult, kas *antud hetkel vajaliku* faili baite sisaldavad vahemälu elemendid on kooskõlalised või mitte, püüdmata samas kindlaks teha *kõiki* mittekooskõlaliseks muutunud elemente.

Andme- või nimevahemälu elemendi kooskõlalisuse kontroll toimub tavaliselt elemendile vastava faili või kataloogifaili viimase modifitseerimise aja järgi, mis kuulub faili atribuutide hulka. Kui kliendimoodul paigutab andme- või nimevahemällu uue elemendi, siis jätab ta meelde ka vastava faili viimase modifitseerimise aja *t*. Kui viidatakse failile, mille sisu täielikult või osaliselt vahemälus hoitakse, omandab kliendimoodul serverprotsessilt viidatud faili atribuudid (näiteks NFSi puhul operatsiooni `getattr` abil). Seejärel võrdleb kliendimoodul faili kohta meeldejäetud aega *t* ja omandatud atribuutide hulgas olevat faili viimase modifitseerimise aega *t'*. Kui $t \neq t'$, kuulutab kliendimoodul kõik faili baite sisaldavad vahemälu elemendid kehtetuks.

Kuigi faili atribuutide hulgas on ka nende viimase modifitseerimise aeg, pole staatusvahemälu puhul mõtet kliendi juhitud valideerimist kasutada. Faili atribuute sisaldava kirje kooskõlalisuse kontrollimiseks tuleks needsamad atribuudid serverprotsessilt omandada, justkui oleks kirje mittekooskõlaline ja kliendimoodul paigutaks faili atribuute uuesti vahemällu. Näiteks NFSi puhul eeldatakse, et staatusvahemälu element on pärast vahemällu paigutamist 60 sekundi jooksul kooskõlaline. Pärast selle ajavahemiku möödumist loetakse element automaatselt kehtetuks, sest selle kooskõlalisust pole mõtet kontrollida [Silb94].

Kliendi juhitud valideerimise kasutamisel on serverarvuti ja sellega ühendatud kohtvõrgu koormus väga suur, sest serverprotsessi poole pöördutakse igal sellisele failile viitamisel, mille sisu parajasti täielikult või osaliselt vahemälus hoitakse. Seetõttu kasutatakse praktilistes hajusates failisüsteemides selle valideerimisstrateegia modifikatsioone - faili baite sisaldavate vahemälu elementide kooskõlalisust kontrollitakse vaid üksikutel sellele failile viitamistel, eeldades, et igal ülejäänud viitamise hetkel on elemendid kooskõlalised. Näiteks NFSi puhul eeldab kliendimoodul, et vahemälu element on pärast viimast kooskõlalisuse kontrolli teatud aja vältel kooskõlaline, ning AFS-1 puhul, et element jääb talle vastava faili avamise ja sulgemise vahel kooskõlaliseks.

3.4.2 Serveri juhitud valideerimine

Kuna kliendi juhitud valideerimise kasutamisel on serverarvuti ja sellega ühendatud kohtvõrgu koormus üsna suur, siis pole seda valideerimisstrateegiat või selle modifikatsioone kasutav hajus failisüsteem eriti skaleeritav [Blaze93].

Kliendi juhitud valideerimise peamine puudus on see, et serverprotsessi poole võidakse pöörduda ka asjatult, sest vahemälu element on tegelikult kooskõlaline. Andmevahemälu korral see enamasti nii ongi, sest kuigi failide sõlmedevaheline jagamine on sage, avatakse selliseid faile harva kirjutamiseks (vt. alapunkt 3.1.1). Serverarvuti ja sellega ühendatud kohtvõrgu koormuse vähendamiseks kasutatakse tavaliselt *serveri juhitud valideerimist* (*server driven validation*). Selle valideerimisstrateegia korral peab serverprotsess meeles, millised kliendimoodulid milliseid andmeid (faile, katalooge või failiatribuute) täielikult või osaliselt vahemälus (andme-, nime- või staatusvahemälus) hoiavad. Kui faili, kataloogi või failiatribuute modifitseeritakse ja tehtud muudatused serverprotsessile nähtavaks saavad, saadab serverprotsess teate kõigile kliendimoodulitele, kes vastavat faili, kataloogi või failiatribuute täielikult või osaliselt vahemälus hoiavad. Talle saadetud teadete põhjal saab kliendimoodul kindlaks teha, millised vahemälu elemendid on mittekooskõlaliseks muutunud.

Serveri juhitud valideerimise eelis on see, et serverarvutile ja võrgule ei asetata mõttetut koormust - teateid saadetakse ainult siis, kui vahemälu mingid elemendid on tõesti mittekooskõlaliseks muutunud. Seda valideerimisstrateegiat kasutavad nii andme- kui staatusvahemälu puhul AFS ja Coda (mõlemas hajusas failisüsteemis on langevad andme- ja nimevahemälu kokku). AFS on mitmete eriteadlaste arvates võrdlemisi skaleeritav just seetõttu, et valideerimisstrateegiana kasutatakse serveri juhitud valideerimist [Blaze93, Nelson88].

Serveri juhitud valideerimise korral tekivad aga järgmised probleemid:

- kui teated faili, kataloogi või failiatribuutide modifitseerimise kohta ei jõua kliendimoodulini (näiteks püsivama võrguvea tõttu, kus ei aita ka teadete korduv edastamine), võivad klientprotsessid väga kaua mittekooskõlaliste vahemälu elementidega töötada. Selle vältimiseks võib kasutada võtet, kus failile, kataloogile või failiatribuutidele viitamisel peab kliendimoodul vastavate vahemälu elementide

kooskõlalisuse kontrollimiseks ise serverprotsessi poole pöörduma, kui teatud aja vältel pole serverprotsessilt ühtegi teadet saabunud. Kui serverprotsessiga kontakti võtta ei õnnestu, kuulutab kliendimoodul vastavad vahemälu elemendid kehtetuks (sellist lahendust kasutab ka AFS).

- serverprotsess peab meeles pidama informatsiooni, millised kliendimoodulid milliseid andmeid vahemälus hoiavad. See võib nõuda palju mälu, mistõttu hajus failisüsteem ei ole populatsiooni suhtes kuigi skaleeritav. Samuti muutub serverprotsessi avariilõpujärgne taaskäivitamine keeruliseks, sest serverprotsess peab kliendimoodulite kohta käiva informatsiooni taastama. Nende probleemide lahendamiseks võib serverprotsess mälu lõppemisel osa informatsiooni kustutada ning teatada vastavatele kliendimoodulitele, et andmeid on modifitseeritud; informatsiooni taastamise lihtsustamiseks võib seda serverarvuti kettal hoida (neid võtteid kasutab ka AFS).

3.4.3 Rentimine

Rentimine on valideerimisstrateegia, kus vahemälu valideerimine toimub *rendilepingute* (*leases*) põhjal, mida serverprotsess kliendimoodulitega sõlmib. Seda valideerimisstrateegiat on oma töös lähemalt kirjeldanud Gray ja Cheriton [Gray90]. Oma käsitluses piirduvad nad andmevahemäluga, kuid rentimist saab kasutada ka nime- ja staatusvahemälu valideerimisstrateegiana. Kui kliendimoodul paigutab andmevahemällu uue elemendi ja elemendile vastava faili kohta rendileping puudub, siis sõlmib serverprotsess kliendimooduliga selle lepingu. Faili puudutav rendileping kehtib teatud aja vältel, mille jooksul kohustub serverprotsess faili modifitseerimisel sellest kliendimoodulile teatama ja lepingu tühistama. Teate saamisel kuulutab kliendimoodul faili baite sisaldavad andmevahemälu elemendid kehtetuks. Kui faili puudutav rendileping on aegunud ja klientprotsess failile viitab, peab kliendimoodul lepingu uuendamiseks uuesti serverprotsessi poole pöörduma. Serverprotsess sõlmib kliendimooduliga uue lepingu ja kui faili on vahepeal modifitseeritud, kuulutab kliendimoodul faili baite sisaldavad andmevahemälu elemendid kehtetuks.

Põhiküsimus on, milline peaks olema rendilepingu kehtimise aeg ehk *lepingu kehtivusaeg* (*lease term*). Null-kehtivusajaga lepingute alusel rentimine kujutab endast

sisuliselt kliendi juhitud valideerimist, lõpmatu kehtivusajaga lepingute alusel rentimine aga serveri juhitud valideerimist. Rentimine, kus iga faili kohta sõlmitakse leping alati nii, et leping kaotab kehtivuse vahetult enne faili modifitseerimist, on optimaalne valideerimisstrateegia [Blaze93]. Sellisel juhul ei pea serverprotsess kliendimoodulitele failide modifitseerimisest teatama ning kliendimoodulid ei pöördu kunagi serverprotsessi poole, kui selleks puudub tegelik vajadus (s.t. siis, kui faili pole modifitseeritud ja faili baite sisaldavad andmevahemälu elemendid on endiselt kooskõlalised). Praktikas seda valideerimisstrateegiat kasutada ei saa, kuna ta nõuab tuleviku etteteadmist.

Lühemate lepingu kehtivusaegade korral on serverarvuti ja sellega ühendatud kohtvõrgu koormus suurem, sest rentimine on sel juhul kliendi juhitud valideerimisele lähedane. Pikemate kehtivusaegade korral peab serverprotsess rohkem lepinguid puudutavat informatsiooni meeles pidama, sest igal ajahetkel kehtib korraga rohkem lepinguid. Seetõttu on hajus failisüsteem populatsiooni suhtes vähem skaleeritav.

Lühemate lepingu kehtivusaegade puhul on serverprotsessi avariilõpujärgne taaskäivitamine lihtsam. Juhul, kui avariilõpu ja taaskäivitamise vaheline ajavahemik on pikem kui suvalise lepingu kehtivusaeg, pole lepinguid puudutavat informatsiooni vaja taastada. Pikemate kehtivusaegade puhul esinevad sellised olukorrad harvemini. Hajusas failisüsteemis AFS, kus valideerimisstrateegiana kasutatakse lõpmatu kehtivusajaga lepingute alusel rentimist, salvestatakse informatsioon selle taastamise lihtsustamiseks serverarvuti kettale.

Null-kehtivusajaga lepingute alusel rentimine on efektiivsem kui nullile lähedase väga lühikese kehtivusajaga lepingute alusel rentimine. Kui sõlmitud leping kaotab kehtivuse enne, kui kliendimoodul jõuab uue elemendi vahemällu paigutada, siis serverarvuti ja sellega ühendatud kohtvõrgu koormus ei vähene. Samas nõuab lepinguid puudutava informatsiooni haldamine täiendavalt serverarvuti protsessoriaega (null-kehtivusajaga lepingute kasutamisel ei pea serverprotsess sellist informatsiooni meeles pidama).

Rentimise kasutamine valideerimisstrateegiana nõuab, et klient- ja serverarvuti kellad oleksid sünkroniseeritud. Kui serverarvuti kell käib kiiremini kui klientarvuti kell, võib serverprotsess teatud hetkel arvata, et leping on kehtivuse kaotanud ning faili modifitseerimisel pole enam vaja kliendimoodulit hoiatada. Nii võib juhtuda, et

klientprotsessid kasutavad teatud aja vältel mittekooskõlalisi vahemälu elemente. Kui klientarvuti kell käib serverarvuti omast kiiremini, pöördub kliendimoodul lepingu uuendamiseks serverprotsessi poole liiga vara. See koormab serverarvutit ja sellega ühendatud kohtvõrku täiendavalt.

Gray ja Cheriton viisid läbi mõned simulatsioonid, et uurida, kui efektiivsed on erinevad lepingu kehtivusajad hajusa failisüsteemi skaleeritavuse suurendamisel (simulatsioonide läbiviimiseks vajalikud lähteandmed koguti hajusa failisüsteemi V installatsioonist). Gray ja Cheriton eeldasid, et kõigi sõlmitud lepingute kehtivusajad on ühepikkused. Lepingu kehtivusaja efektiivsust hindasid nad teadete arvu järgi, mida serverprotsess peab andmevahemälude valideerimisel saatma või vastu võtma. Nad leidsid, et juba 10 sekundi pikkuste kehtivusaegadega lepingute kasutamisel väheneb saadetud ja vastuvõetud teadete arv ning seega ka serverarvuti ja sellega ühendatud kohtvõrgu koormus tunduvalt (võrreldes juhuga, kus rentimine toimub null-kehtivusajaga lepingute alusel); pole erilist vahet, kas lepingud kehtivad 30 sekundit või kauem [Gray90]. See tulemus on võrdlemisi loomulik, sest failide kasutamine on ajaliselt lokaalne (vt. alapunkt 3.1.1) - tüüpiliselt kasutab klientprotsess faili lühema ajavahemiku jooksul palju kordi, millele järgneb pikem ajavahemik, kus ta faili üldse ei kasuta. Kui faili hakatakse aktiivselt kasutama, sõlmitakse faili puudutav leping või uuendatakse seda. Ka mõnekümne sekundi pikkuse kehtivusajaga lepingu puhul on tõenäoline, et selle kehtimise ajal viidatakse failile palju kordi; faili aktiivse kasutamise periood võib osutuda lühemaks kui lepingu kehtivusaeg. See tähendab, et suurem osa failile viitamistest on sellised, kus lepingu uuendamiseks pole vaja serverprotsessi poole pöörduda.

3.5 Vahemälu füüsilised parameetrid

3.5.1 Vahemälu suurus

Mida suurem on vahemälu, seda rohkem andmeid sinna mahub ning seda suurem on vahemälu tabamustegur ja väiksem selle veategur. Hajusa failisüsteemi vahemälude disainimisel pole otstarbekas vahemälude suurusi fikseerida. Erinevatel aastatel tehtud

uuringute käigus on näiteks leitud, et fikseeritud suurusega andmevahemälu veategur kasvab aja möödudes [Oust85, Baker91].

Kui vahemälu suurus oleks fikseeritud, siis vähendaks see hajusa failisüsteemi skaleeritavust - pole kindel, et mõne aasta möödudes on vahemälu tabamustegur sama kõrge ning vahemälu pole vaja suurendada. Kettal paiknevate vahemälude suurust hajusa failisüsteemi vahemälude disainimisel tavaliselt kindlaks ei määrata. Sellise vahemälu suurus on piiratud vaid kettapartitsiooni suurusega, kus vahemälu sisu hoitakse; vahemälu suurendamiseks suurendatakse vastavat kettapartitsiooni.

Klient- või serverarvuti operatiivmälus paiknevate kliendi või serveri vahemälude korral kasutatakse tavaliselt skeemi, kus vahemälu jaoks eraldatakse teatud protsent klient- või serverarvutisse installeeritud operatiivmälust. Kuna aja möödudes hajusa failisüsteemi sõlmedesse installeeritud operatiivmälu hulk üldiselt kasvab, siis aja möödudes kasvab ka vahemälu suurus. See lahendus on serveri vahemälude puhul levinuim - paljudes hajusates failisüsteemides kasutatakse serveri vahemäluna serverarvuti operatsioonisüsteemi poolt loodud kettavahemälu (vt. lk. 57), mille suurus tavaliselt kirjeldatud skeemi järgi määrataksegi. Näiteks UNIX eraldab kettavahemälule tavaliselt 10% arvuti operatiivmälust [Baker91].

See meetod on teatud mõttes *staatiline*, sest klient- või serverarvuti töö käigus vahemälu suurus ei muutu. Samas võib juhtuda, et mõnel ajahetkel kasutavad hajusa failisüsteemi sõlmes töötavad protsessid faile väga intensiivselt, vajamata selleks kuigi palju virtuaalmälu. Sellisel juhul oleks otstarbekas vahemälu suurendada, vähendades virtuaalmälumoodulile eraldatud operatiivmälu hulka. Võib ka juhtuda, et mõnel ajahetkel vajavad hajusa failisüsteemi sõlmes töötavad protsessid palju virtuaalmälu, kuid ei kasuta eriti intensiivselt faile. Sellisel juhul on mõtet vahemälu suurust vähendada, loovutades osa operatiivmälust virtuaalmälumoodulile.

Hajusas failisüsteemis Sprite jagatakse hajusa failisüsteemi sõlme operatiivmälu vastavalt tekkinud vajadustele vahemälu ja virtuaalmälumooduli vahel *dünaamiliselt*. Jagamisalgoritmi töötas 1987. aastal välja Nelson [Nelson87]. Algoritm opereerib operatiivmälu lehekülgedega, võttes vajadusel ühelt osapoolelt lehekülje ja andes selle teisele. Sprite on seni ainuke tuntum hajus failisüsteem, kus kliendi- või serveri vahemälu suurus võib klient- või serverarvuti töö käigus muutuda [Silb94]. Hajusas failisüsteemis Sprite on nii kliendi kui serveri vahemälu elementideks blokid; bloki

suurus on valitud nii, et operatiivmälu lehekülje suurus on blokisuuruse kordne (näiteks Sprite'i Sun-3 arhitektuuril töötava versiooni korral on kliendi ja serveri vahemälu elementideks 4Kb suurused blokid; lehekülje suuruseks on 8Kb [Nelson87]). Järgnevas kirjeldatakse, kuidas jagamisalgoritm klientarvuti operatiivmälu puhul töötab (serverarvuti korral on algoritm analoogiline).

Jagamisalgoritm kasutab ära asjaolu, et nii kliendimoodul kui virtuaalmälumoodul rakendavad pärast neile eraldatud operatiivmälu osa täissaamist uutele andmetele ruumi tegemiseks LRU-algoritmi - viimati kõige kauem aega tagasi kasutatud ehk *vanim* blokk või virtuaalmälu lehekülg asendatakse uue bloki või virtuaalmälu leheküljega. Kui ühele moodulile eraldatud operatiivmälu osa täis saab ning moodul vajab uute andmete jaoks ruumi, leiab ta endale kuuluva vanima lehekülje *vanuse* (s.t. kui pikka aega tagasi seda lehekülge viimati kasutati) ning võrdleb seda teisele moodulile kuuluva vanima lehekülje vanusega. Kui teise mooduli vanim lehekülg on vanem, võetakse see teiselt moodulilt enda mäluvajaduste rahuldamiseks, vastasel korral kasutatakse uute andmete operatiivmällu paigutamiseks endale kuuluvat vanimat lehekülge. Tekib küsimus, kuidas leida kliendimoodulile kuuluva operatiivmälu lehekülje vanust, kui bloki suurus on lehekülje suurusest väiksem ja lehekülg sisaldab mitut blokki. Sprite'i korral loetakse lehekülje vanuseks sellel leheküljel asuva noorima bloki vanust [Nelson87]. Kui kliendimoodulil õnnestub virtuaalmälumoodulile kuulunud lehekülg endale saada, siis eraldatakse talle operatiivmälu mitme bloki jaoks, kuigi tegelikult oleks vaja läinud ruumi üheainsa uue bloki mahutamiseks.

Kui seda algoritmi Sprite'i puhul kasutada, tekivad mõned probleemid [Nelson87]. Hajusas failisüsteemis Sprite hoitakse operatiivmällu mittemahtuvaid virtuaalmälu lehekülgi serverarvutite ketastel asuvates *saalefailides* (*swap files*)[14], mistõttu virtuaalmälumoodul on failiteenuse klient. Seetõttu peaks kliendimoodul virtuaalmälu lehekülje operatiivmällu paigutamisel leheküljele vastavad saalefaili blokid ka kliendi vahemällu paigutama, sest neile blokkidele viidati. Operatiivmälu sellise raiskamise vältimiseks ei lubata saalefailide blokke kliendi vahemällu paigutada (küll aga hoitakse saalefailide blokke serveri vahemälus, et *lehekülgede saalimist* (*paging*) kiirendada [Nelson87]). Probleem säilib juhu jaoks, kus virtuaalmälumooduli poolt

[14] Peamiselt sellepärast, et hõlbustada protsesside migratsiooni (protsesside liikumist ühest hajussüsteemi sõlmest teise) [Silb94].

loetud leheküljele vastavad blokid on kliendi vahemälus juba olemas. Näiteks programmifailist koodi operatiivmällu laadimisel võib juhtuda, et lehekülgedele vastavad blokid on ka vahemälus, sest programm kompileeriti hiljuti. Sellisel juhul loeb virtuaalmälumoodul vajalikud virtuaalmälu leheküljed otse kliendi vahemälust, lehekülgedele vastavad blokid kuulutatakse aga vahemälu vanimateks blokkideks, et nad võimalikult kiiresti vahemälust välja jäetaks.

On selge, et võrreldes staatilise meetodiga, kus vahemälu suurus on fikseeritud, töötab kirjeldatud dünaamiline meetod palju paremini juhtudel, kus enamik hajusa failisüsteemi sõlmes töötavaid protsesse kasutab intensiivselt kas faile või virtuaalmälu, sest vastavalt vajadusele võib vahemälule või virtuaalmälumoodulile eraldatud operatiivmälu hulk üsna suureks kasvada. Hajusas failisüsteemis Sprite on tüüpiline, et serveri vahemälule eraldatakse peaaegu kogu operatiivmälu, sest serverarvutis töötav serverprotsess kasutab serverarvuti kettal asuvaid faile väga intensiivselt, kuid nõuab oma tööks vähe virtuaalmälu [Baker91].

Ei ole aga selge, kui efektiivne on dünaamiline meetod võrreldes staatilisega neil juhtudel, kus klientprotsessid kasutavad intensiivselt nii paljusid failiblokke kui ka paljusid virtuaalmälu lehekülgi. Nelson koostas enda väljatöötatud jagamisalgoritmi efektiivsuse hindamiseks erilise *ECD-testi* (*edit-compile-debug benchmark*), mille täitmine nõuab palju virtuaalmälu ning mille täitmise käigus loetakse failidest ja kirjutatakse failidesse suurem hulk baite (vt. Tabel 3.7). Algoritmi efektiivsust hindas ta testi täitmise aja (mida lühem, seda parem) ning serverarvuti protsessori koormuse (mida väiksem, seda parem) järgi [Nelson88].

Testi komponent	**Komponendi kirjeldus**	**Failidest loetud ja sinna kirjutatud baitide hulk**	**Virtuaalmälu hulk, mida komponent vajab**
Edit	2500-realise faili redigeerimine	70 Kb	560Kb
Compile	Virtuaalmälumooduli kompileerimine	800Kb	1Mb
Link	Operatsioonisüsteemi tuuma linkimine	8Mb	3Mb
Debug	Operatsioonisüsteemi tuuma silumine	4Mb	8,5Mb
Keskkond	***X Window System***		***5Mb***

Tabel 3.7 ECD-test.

Nelson tegi kindlaks, et staatilise meetodi puhul täidetakse test kõige kiiremini ning koormatakse serverarvuti protsessorit kõige vähem siis, kui vahemälu on väike (umbes 0,5-1Mb suurune) [Nelson88]. Seetõttu võrdles ta jagamisalgoritmi staatilise meetodi kahe erijuhuga, kus vahemälu on vastavalt 0,5Mb ja 1Mb suurune. Nelsoni mõõtmistulemused näitavad, et kui klientprotsessid kasutavad intensiivselt faile ja nõuavad samuti palju virtuaalmälu, ei ole tema poolt väljatöötatud jagamisalgoritm vähem efektiivne kui staatiline meetod (vt. Tabel 3.8 ja Tabel 3.9).

Operatiiv-mälu hulk / *Vahemälu suurus*	**10Mb**	**11Mb**	**12Mb**	**14Mb**	**16Mb**
0,5Mb	951	793	768	714	667
1Mb	1899	850	783	740	690
muutuv	942	840	787	692	663

Tabel 3.8 ECD-testi täitmise aeg sekundites sõltuvalt kliendi vahemälu suurusest ja klientarvutisse installeeritud operatiivmälu hulgast.

Operatiiv-mälu hulk / *Vahemälu suurus*	**10Mb**	**11Mb**	**12Mb**	**14Mb**	**16Mb**
0,5Mb	23,87%	21,82%	21,18%	19,80%	17,40%
1Mb	36,73%	21,57%	21,47%	19,28%	18,49%
muutuv	23,37%	21,96%	20,86%	18,99%	18,20%

Tabel 3.9 Serverarvuti protsessori utiliseeritus ECD-testi täitmisel sõltuvalt kliendi vahemälu suurusest ja klientarvutisse installeeritud operatiivmälu hulgast.

Nelson mõõtis sedagi, kui suurtes piirides kliendi vahemälu suurus muutub; samuti mõõtis ta, mitmel korral üks moodul uue lehekülje omandamiseks teise mooduli poole pöördub ning mitmel korral pöördumine rahuldatakse (s.t. vanimate lehekülgede võrdlusel osutub teisele moodulile kuuluv lehekülg vanemaks) (vt. Tabel 3.10).

Operatiiv-mälu hulk	Vahemälu suuruse muutumine		Kliendimooduli pöördumised		Virtuaalmälumooduli pöördumised	
	Vahemälu min. suurus	Vahemälu max. suurus	Pöördumiste arv	Neist rahuldatud	Pöördumiste arv	Neist rahuldatud
10Mb	0,25Mb	5,6Mb	8125	1810	2942	1846
11Mb	0,25Mb	6,4Mb	7105	1889	2610	1967
12Mb	0,25Mb	6,9Mb	5840	1964	2555	2075
14Mb	0,25Mb	8,7Mb	4012	1957	2669	2162
16Mb	0,34Mb	8,8Mb	3652	1937	2629	2229

Tabel 3.10 ECD-testi täitmise ajal toimuvad sündmused, kui kasutatakse Nelsoni jagamisalgoritmi.

Nelsoni mõõtmistulemused näitavad, et vahemälu suurus võib klientarvuti töö käigus väga palju muutuda. Mida rohkem operatiivmälu klientarvutisse installeeritud on, seda suuremates piirides vahemälu suurus kõigub. Operatiivmälu hulga kasvades kasvab ka teise mooduli poolt rahuldatud pöördumiste arv. Samas õnnestub virtuaalmälumoodulil kliendimoodulilt palju sagedamini lehekülg ära võtta kui vastupidi. Nelson järeldas sellest, et virtuaalmälumoodul kasutab oma lehekülgi palju aktiivsemalt kui kliendimoodul, sest virtuaalmälumooduli vanim lehekülg osutub enamasti kliendimooduli vanimast leheküljest nooremaks. Samuti on näha, et operatiivmälu hulga kasvades väheneb küll kliendimooduli pöördumiste arv, kuid virtuaalmälumooduli pöördumiste arv ei vähene. Nelson järeldas, et kui protsessid kasutavad intensiivselt paljusid failiblokke ning paljusid virtuaalmälu lehekülgi, siis on virtuaalmälumooduli mäluvajadused palju suuremad kui kliendimoodulil [Nelson88]. (Ilmselt sellepärast ongi staatilise meetodi kasutamisel ECD-testi puhul väike kliendi vahemälu kõige efektiivsem.)

Mõõtmiste tulemused ning fakt, et interaktiivsete programmide reaktsiooniajad sõltuvad palju enam virtuaalmälusüsteemist kui failisüsteemist, viisid Nelsoni mõttele, et virtuaalmälumoodulile tuleks operatiivmälu kasutamisel anda prioriteet. Selleks täiendas ta oma jagamisalgoritmi nii, et enne erinevatele moodulitele kuuluvate lehekülgede vanuste võrdlemist vähendatakse virtuaalmälumoodulile kuuluva lehekülje vanust 20 minuti võrra. Mõõtmiste tulemused näitasid, et jagamisalgoritmi modifikatsiooni kasutamisel ECD-testi täitmise aeg ning serverarvuti protsessori

koormus märkimisväärselt ei suurenenud; samas vähenesid interaktiivsete programmide reaktsiooniajad [Nelson88].

3.5.2 Vahemälu element ja vahemälu asukoht

Vahemälude disaini efektiivsust mõjutab ka see, millised on vahemälude elemendid.

Andmevahemälu elementideks võivad olla failiblokid või terved failid. Kui andmevahemälu elementideks on blokid, kasutatakse tavaliselt võrdlemisi suuri blokke. Näiteks NFSi korral on andmevahemälu elementideks 8Kb suurused ja Sprite'i korral 4Kb suurused blokid. Suuremad blokid on populaarsed eestkätt seetõttu, et nende kasutamine toob sisuliselt kaasa nõudmisel toimuva eellugemise (sest faile kasutatakse peaaegu alati järjestikuselt, vt. alapunkt 3.1.1). Kui andmevahemälu elementideks on failid, välditakse sellega lisaks faili kasutamise ajal toimuvaid pöördumisi serverprotsessi poole (lugemaks faili blokke, mida andmevahemälus ei ole), sest klientprotsessid kasutavad enamasti kogu faili sisu (vt. alapunkt 3.1.1). Failide kui andmevahemälu elementide kasuks kõneleb ka see, et *n* bloki lugemine (serveri vahemälust või serverarvuti kettalt) üheainsa operatsiooniga koormab serverarvuti protsessorit ja ketast vähem kui nende blokkide lugemine *n* eraldi operatsiooniga [Sat93, Thom87].

Nimevahemälu elementideks võivad olla üksikud kataloogifailide kirjed, kataloogifailide blokid või terved kataloogifailid. Shirriff ja Ousterhout leidsid 1992. aastal, et kataloogifailid on keskmiselt 1,1Kb suurused. Kuigi leidub ka tunduvalt suuremaid kataloogifaile, ei ületa nende suurus mõnda kilobaiti [Shir92]. See tähendab, et enamasti koosneb kataloogifail vaid ühestainsast blokist ning pole erilist vahet, kas nimevahemälu elementideks on kataloogifailid või kataloogifailide blokid.

Kui nimevahemälu elementideks on kataloogifailide üksikud kirjed, on kliendimoodulil nimevahemälu sisu põhjal võimatu kindlaks teha, kas faili tekstiline nimi on korrektne või mitte (s.t. kas nimele vastab mingi fail või mitte). Shirriff ja Ousterhout leidsid, et umbes 20% kõigist failinimedest, mis kliendimoodulile interpreteerimiseks esitatakse, on mittekorrektsed; samuti leidsid nad, et kui klientprotsess viitab mingi kataloogifaili kirjele, siis lühikese aja pärast viitab klientprotsess selle kataloogifaili mõnele teisele kirjele [Shir92]. Seetõttu on

kataloogifaile või kataloogifailide blokke sisaldava nimevahemälu tabamustegur kõrgem kui kirjeid sisaldava nimevahemälu oma.

Andme- või nimevahemälu võidakse paigutada klientarvuti kettale, sest kettal paiknev vahemälu mahutab tunduvalt enam andmeid kui operatiivmälus paiknev. Tavaliselt paigutatakse vahemälu klientarvuti kettale siis, kui vahemälu elementideks on failid. Kuigi enamasti kasutatakse väikeseid faile, leidub kasutatavate failide hulgas ka üksikuid suuri faile, mis operatiivmälus paiknevasse väiksemasse vahemällu ära ei mahuks. Kettal asuva vahemälu eeliseks on veel see, et selle sisu ei lähe klientarvuti töö avariilõpu korral kaduma. Kui klientarvuti taaskäivitatakse, on vahemälu sisu säilinud. Vahemälu paigutamine operatiivmällu võimaldab aga klientarvutitel olla ilma lokaalse kettata. Samuti saab operatiivmälus paiknevast vahemälust andmeid kiiremini kätte kui kettal paiknevast.

Mitmed hajusate failisüsteemide uurijad on leidnud, et klientprotsessid pöörduvad küllalt tihti süsteemifunktsiooni `stat` poole, kasutades seda põhiliselt faili olemasolu kontrollimiseks [Howard88]. Et kiirendada selle süsteemifunktsiooni täitmist, paigutatakse staatusvahemälu tavaliselt klientarvuti operatiivmällu. Staatusvahemälu elementideks on failide atribuute sisaldavad kirjed. Nende kirjete suurus on väga väike (tavaliselt mitte üle 100-120 baidi), mistõttu isegi sadade failide atribuute mahutav staatusvahemälu pole kuigi suur ning nõuab suhteliselt vähe (40-50Kb) operatiivmälu.

Serveri vahemälu paikneb alati serverarvuti operatiivmälus ning selle elementideks on kettablokid. John Ousterhout mõõtis 1985. aastal, kuidas erinevad kettabloki suurused serverarvuti ketta koormust mõjutavad [Oust85]. Ketta koormust hindas ta bloki kettalt lugemise ja kettale kirjutamise operatsioonide arvu järgi. Serveri kirjutamisstrateegiaks oli *FD-strateegia* (*full-delay policy*), mille kasutamisel kirjutatakse modifitseeritud kettablokk serverarvuti kettale alles siis, kui ta vahemälust välja jäetakse. Mõõtmistulemused näitasid, et suuremate kettablokkide kasutamine vähendab serverarvuti ketta koormust tunduvalt (vt. Tabel 3.11).

Kettabloki suurus	Vahemälu puudub	400Kb vahemälu	2Mb vahemälu	4Mb vahemälu	8Mb vahemälu
1Kb	1432179	562492	280056	227299	194724
2Kb	925934	365806	165312	129654	110369
4Kb	623573	268864	110182	84164	69651
8Kb	527634	**259941**	**90539**	65302	51635
16Kb	481052	280068	103223	**63330**	**47626**
32Kb	461976	307002	156523	82350	51883

Tabel 3.11 Kettabloki suuruse mõju serverarvuti ketta koormusele.

Suuremate blokkide efektiivsust saab põhjendada asjaoluga, et nende kasutamine toob sisuliselt kaasa nõudmisel toimuva eellugemise, kuna klientprotsessid opereerivad failidega peaaegu alati järjestikuselt (vt. alapunkt 3.1.1). Kui kettabloki suurus on ületanud teatud piiri, hakkab kettapöörduste arv jälle kasvama, sest fikseeritud suurusega vahemälu korral tähendab bloki suuruse kasv, et vahemällu mahub vähem blokke. Seetõttu serveri vahemälu veategur tõuseb ja serverarvuti ketta koormus suureneb. Parim lahendus on kasutada suuri kettablokke koos suure serveri vahemäluga.

3.6 Alternatiivsed vahemälustrateegiad

Klient-server mudelil põhinevas hajusas failisüsteemis kasutatakse vahemälustrateegiat, kus vajalikke andmeid püütakse kõigepealt kliendi vahemälust hankida, siis serveri vahemälust ning lõpuks serverarvuti kettalt (vt. lk. 57). Paraku pole klient-server mudelil põhinevad hajusad failisüsteemid kuigi skaleeritavad, sest sellise vahemälustrateegia kasutamisel osutuvad serverarvuti ja sellega ühendatud kohtvõrk hajusa failisüsteemi kasvu takistavateks pudelikaelteks. Juhul, kui vajalikke andmeid kliendi vahemälus ei leidu, pöördub kliendimoodul alati serverprotsessi poole. Hajusa failisüsteemi laienemisel jõutakse lõpuks hetkeni, kus serverarvuti pole kõigi klientprotsesside teenindamiseks enam piisavalt võimas ning serverarvutiga ühendatud kohtvõrk on pidevalt üle koormatud.

Mõned hajusate failisüsteemide uurijad on välja töötanud teistsuguseid vahemälustrateegiaid, mis tagavad hajusa failisüsteemi suurema skaleeritavuse, kuid

mille kasutamine tähendab, et süsteem pole enam klient-server mudelile üles ehitatud. Selle punkti alapunktides vaadeldaksegi mõningaid selliseid *alternatiivseid* vahemälustrateegiaid. Enamike alternatiivsete vahemälustrateegiate korral (välja arvatud alapunktis 3.6.1 vaadeldav strateegia) teenindavad klientprotsesse peale serverprotsesside ka teiste klientarvutite juures asuvad kliendimoodulid. See tähendab, et klient-server mudeli korral serverarvutile ja sellega ühendatud kohtvõrgule asetunud koormus jaotub nüüd paljude hajusasse failisüsteemi kuuluvate sõlmede ja kohtvõrkude vahel. Kõigi nende vahemälustrateegiate korral serverprotsessi ja serverarvuti mõisted siiski säilivad, sest nagu edaspidi näeme, mängib serverprotsess vaatamata oma tähtsuse vähenemisele hajusas failisüsteemis siiski koordineerivat osa.

Alternatiivsete vahemälustrateegiate efektiivsuse hindamiseks kasutavad nende autorid testimist ja simulatsioone, mõõtes, kui palju alternatiivse strateegia puhul serverarvuti ja sellega ühendatud kohtvõrgu koormus väheneb. Nii saab kindlaks teha, kui palju alternatiivse vahemälustrateegia kasutamine hajusa failisüsteemi skaleeritavust suurendab. Veendumaks, et alternatiivse strateegia kasutamisega ei teki hajusasse failisüsteemi uusi pudelikaelu, mõõdetakse harilikult ka kõigi ülejäänud hajusasse failisüsteemi kuuluvate sõlmede ja kohtvõrkude koormust.

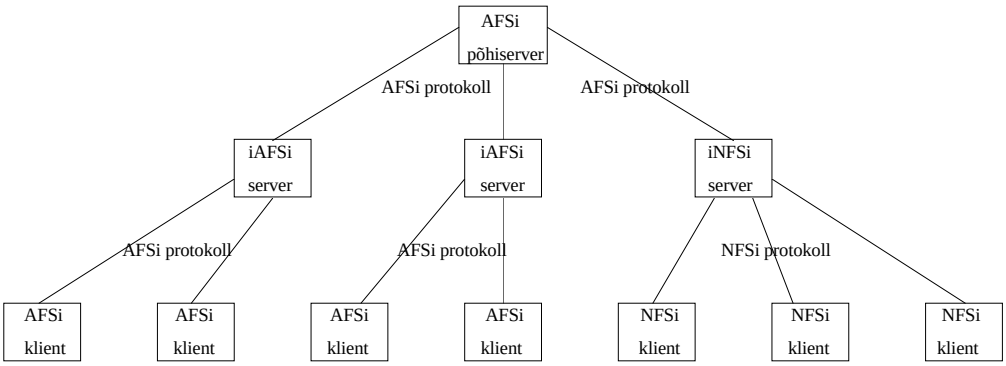
3.6.1 “Serverite staatilise hierarhia” vahemälustrateegia

Muntz ja Honeyman tegid 1992. aastal ettepaneku kasutada vahemälustrateegiat, kus andmeid püütaks kõigepealt kliendi vahemälust hankida, siis *vahepealsest vahemälust* (*intermediate cache*), seejärel serveri vahemälust ning lõpuks serverarvuti kettalt [Muntz92]. Selle vahemälustrateegia kasutamisel on hajusa failisüsteemi sõlmedeks peale klient- ja serverarvutite veel *vahepealsed serverid* (*intermediate servers*), mille juures vahepealsed vahemälud asuvad. Et serverarvutil ja vahepealsel serveril vahet teha, nimetatakse serverarvutit *põhiserveriks* (*principal server*).

Kui vajalikke andmeid kliendi vahemälus pole, pöördub kliendimoodul nende omandamiseks vahepealse serveri juures töötava serverprotsessi ehk *vahepealse serverprotsessi* poole. Kui andmeid pole ka vahepealses vahemälus, pöördub vahepealne serverprotsess omakorda põhiserveri juures töötava serverprotsessi ehk *põhiserverprotsessi* poole. Enne omandatud andmete kliendimoodulile tagastamist

salvestab vahepealne serverprotsess need vahepealsesse vahemällu, et järgmisi samu andmeid puudutavaid nõudeid oleks võimalik kiiremini rahuldada.

Muntz ja Honeyman realiseerisid seda vahemälustrateegiat kasutava hajusa failisüsteemi, võttes aluseks AFSi [Muntz92]. Hajusa failisüsteemi serverarvutid on varustatud AFSi serveritarkvaraga, vahepealsed serverprotsessid suhtlevad põhiserverprotsessidega AFSi protokolli kasutades (hajusa failisüsteemi osa, mis ei sisalda klientarvuteid, kujutab endast sisuliselt harilikku AFSi installatsiooni). Osa vahepealsete serverite juures töötavaid serverprotsesse suhtleb kliendimoodulitega AFSi protokolli abil, osa aga mõne teise (näiteks NFSi) protokolli abil (vt. Joonis 3.10). See võimaldab failiteenust kasutada ka neis klientarvutites töötavatel klientprotsessidel, mis pole AFSi klienditarkvaraga varustatud. Vahepealset serverit, mille juures töötav serverprotsess suhtleb kliendimoodulitega AFSi protokolli kasutades, nimetatakse *iAFSi serveriks* (*intermediate AFS server*).

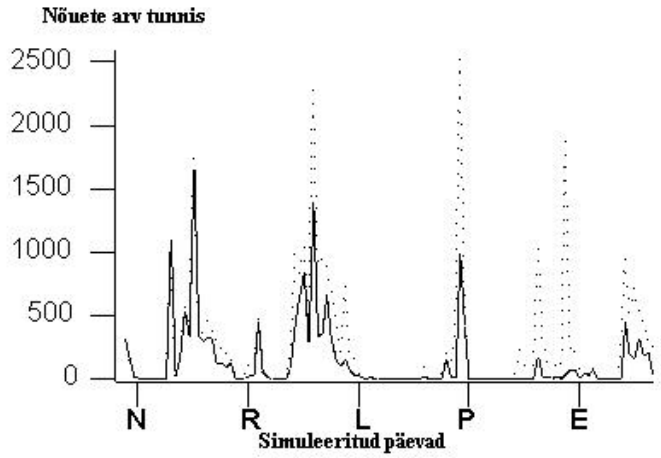


Joonis 3.10 “Serverite staatilise hierarhia” vahemälustrateegiat kasutav hajus failisüsteem.

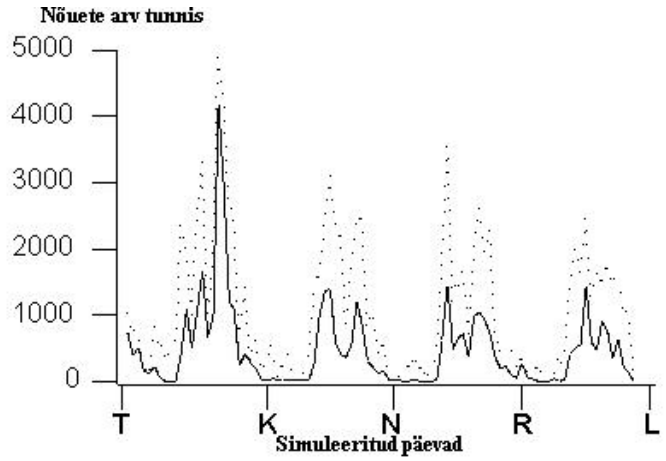
Muntz ja Honeyman uurisid ka, kui efektiivne nende väljatöötatud vahemälustrateegia hajusa failisüsteemi skaleeritavuse suurendamisel on [Muntz92]. Selleks viisid nad läbi mitu simulatsiooni, eeldades, et hajusas failisüsteemis on 1 iAFSi server ja 1 põhiserver ning kõik klientarvutid on varustatud AFSi klienditarkvaraga. Samuti eeldasid Muntz ja Honeyman, et vahepealne vahemälu koosneb ainult andmevahemälust (s.t. seal ei hoita kataloogide sisu või failiatribuute) ning on lõpmatu suurusega; kliendi vahemälud on 20Mb suurused. Vahemälustrateegia efektiivsuse hindamiseks mõõtsid nad, mitu korda vahepealse serverprotsessi ja põhiserverprotsessi poole faili omandamise

nõudega pöördutakse (faili omandamiseks kasutab kliendimoodul või vahepealne serverprotsess operatsiooni `Fetch`). Simulatsioonide läbiviimiseks kasutati lähteandmeid, mis olid kogutud 115 sõlmest koosnevast hajusa failisüsteemi Topaz installatsioonist (4 päeva jooksul) ning 49 klient- ja 4 serverarvutiga AFSi installatsioonist (4,8 päeva jooksul).

Mõõtmistulemused näitavad, et vahepealne vahemälu vähendab põhiserverprotsessile saadetud nõuete arvu (vt. Joonis 3.11 ja Joonis 3.12), seda eriti aga siis, kui vahepealne vahemälu on *soojenenud* (s.t. vahemälu on mõnda aega kasutatud). Selle põhjuseks on asjaolu, et võrdlemisi suur osa failide avamistest on sellised, kus avatavat faili on juba kasutanud teiste sõlmede juures töötavad (või töötanud) klientprotsessid (vt. alapunkt 3.1.1). Kui klientprotsess sellise faili avab ning seda kliendi vahemälus ei ole, on see suure tõenäosusega vahepealses vahemälus, mistõttu faili omandamiseks pole vaja põhiserverprotsessi poole pöörduda.



Joonis 3.11 Vahepealse serverprotsessini (tähistatud punktiiriga) ja põhiserverprotsessini (tähistatud pideva joonega) jõudnud nõuete arv tunnis (AFSi installatsioonist kogutud lähteandmete korral).



Joonis 3.12 Vahepealse serverprotsessini (tähistatud punktiiriga) ja põhiserverprotsessini (tähistatud pideva joonega) jõudnud nõuete arv tunnis (Topazi installatsioonist kogutud lähteandmete korral).

Muntz ja Honeyman märkasid ka, et vahepealse vahemälu tabamusteguri ja ajaühikus vahepealsele serverprotsessile saadetud nõuete arvu vahel on tugev sõltuvus - kui ajaühikus saabub vahepealsele serverprotsessile rohkem nõudeid, tõuseb kohe ka vahepealse vahemälu tabamustegur [Muntz92]. Kui vahepealse vahemälu soojenemisajaks lugeda pool simuleeritud ajast ja pärast seda saabub vahepealsele serverprotsessile 1 tunni pikkuse ajavahemiku vältel keskmisest tunduvalt enam nõudeid, siis põhiserverprotsessile edastatakse neist mitte rohkem kui pooled (vt. Joonis 3.11 ja Joonis 3.12). Põhiserver ja sellega ühendatud kohtvõrk pole kunagi väga koormatud, sest kriitilistel hetkedel on vahepealse vahemälu tabamustegur võrdlemisi kõrge.

See näitab, et Muntzi ja Honeymani väljatöötatud vahemälustrateegia on võrdlemisi efektiivne - vahepealse vahemälu kasutamine vähendab serverarvuti ja sellega ühendatud kohtvõrgu koormust ning suurendab seega hajusa failisüsteemi skaleeritavust.

3.6.2 "Klientide dünaamiliste hierarhiate" vahemälustrateegia

Muntzi ja Honeymani poolt väljatöötatud vahemälustrateegia korral on võrdlemisi ebamugav on kasutada faile, millega opereerivad ainult ühes sõlmes töötavad

klientprotsessid. Kui antud sõlmes töötav klientprotsess sellise faili avab ja seda (või selle blokke) kliendi vahemälus pole, pole seda faili (või selle blokke) tõenäoliselt ka vahepealses vahemälus. See tähendab, et failis sisalduvate andmete omandamiseks tuleb ikkagi serverprotsessi poole pöörduda, kuid sellele kulub kokkuvõttes rohkem aega kui traditsioonilise vahemälustrateegia korral, sest kliendimooduli nõue käib vahepealse serverprotsessi käest läbi.

See puudus ajendas Matthew Blaze'i uut vahemälustrateegiat välja töötama [Blaze93]. Ta eeldas, et kliendi vahemälu koosneb ainult andmevahemälust, mille elementideks on failid. Kui klientprotsess faili avab, püütakse seda kõigepealt kliendi vahemälust hankida. Kui vajalikku faili kliendi vahemälus ei ole ja faili jagavad vähem kui Δ erinevas sõlmes töötavad klientprotsessid, püütakse seda seejärel serveri vahemälust ning lõpuks serverarvuti kettalt hankida; kui faili jagavad Δ või enamas sõlmes töötavad klientprotsessid, omandatakse vajalik fail mõne teise klientarvuti juures asuvast kliendi vahemälust.

Järgnevas on toodud selle vahemälustrateegia täpne kirjeldus [Blaze93]:

- Iga klientarvuti juures asub *nimevahemälu* (*name cache*)[15]. Nimevahemälu elementideks on kirjed, millest igaüks koosneb faili nimest, *ülemuse* (*parent*) nimest ja *alluvate nimekirjast*, mis sisaldab kuni Δ *alluva* (*child*) nimesid. Ülemuseks võib olla serverprotsess või mõne teise klientarvuti juures asuv kliendimoodul, alluvaks on alati mõne teise klientarvuti juures asuv kliendimoodul. Nimevahemälu peab olema vähemalt nii suur, et mahutada kirjeid kõigi kliendi vahemälus asuvate failide kohta.
- Serverarvuti juures paikneb samuti nimevahemälu, mille elementideks olevatest kirjetest igaüks koosneb faili nimest ja kuni Δ alluva nimesid sisaldavast alluvate nimekirjast. Nimevahemällu lisatakse uus mingi faili kohta käiv kirje siis, kui faili omandamiseks esimest korda serverprotsessi poole pöördutakse või kui fail luuakse. Mõlemal juhul kantakse algselt tühja alluvate nimekirja serverprotsessi poole pöördunud kliendimooduli nimi.
- Kui klientprotsess avab faili, püütakse seda kõigepealt kliendi vahemälust leida. Kui faili kliendi vahemälus pole, otsitakse nimevahemälust faili kohta käivat kirjet. Kui

[15] Blaze kasutab oma töös [Blaze93] terminit *nimevahemälu* teises tähenduses, kui siiani käesolevas töös on kasutatud. Töö selles osas (alapunkt 3.6.2) on edaspidi järgitud Blaze'i eeskuju.

seal leidub faili kohta kirje, pöördutakse faili omandamiseks ülemuse poole, vastasel korral serverprotsessi poole. Serverprotsessilt saadud vastus sisaldab nõutud faili või nende kliendimoodulite nimekirja, kes antud faili juba vahemälus hoiavad. Kui vastuseks on nimekiri, valitakse sealt sobiv kliendimoodul ja pöördutakse faili omandamiseks selle poole. Kui vastuseks on jälle nimekiri, valitakse sealt uus sobiv kliendimoodul ja pöördutakse selle poole jne., kuni vastuseks saadakse fail. Seejärel paigutatakse nimevahemällu antud faili kohta käiv kirje, kus ülemusena märgitakse osapool (kliendimoodul või serverprotsess), kellelt fail lõpuks omandati.

- Kui kliendimoodulile või serverprotsessile saabub nõue mingi faili omandamiseks, siis otsitakse nimevahemälust selle faili kohta käiv kirje ja kontrollitakse, kas nõudja on juba alluvate nimekirjas. Kui see nii on, saadetakse nõudjale vajalik fail. Kui see nii ei ole ja alluvate nimekirjas on Δ nime, siis saadetakse nõudjale vastuseks alluvate nimekiri. Kui alluvate nimekirjas on vähem kui Δ nime, saadetakse vastuseks fail ja nõudja nimi lisatakse alluvate nimekirjale.
- Kui kliendimoodulile saabub nõue mingi faili omandamiseks ja kliendimoodul on kohustatud selle faili saatma (sest nõudja on juba alluvate nimekirjas või alluvate nimekirjas on vähem kui Δ nime), kuid fail on juba kliendi vahemälust välja jäetud, pöördub kliendimoodul faili kohta käiva nimevahemälu kirje põhjal leitud ülemuse poole ja omandab vajaliku faili ülemuselt, seejärel aga edastab faili nõudjale. Kui nimevahemälus antud faili kohta enam kirjet ei ole, omandab kliendimoodul vajaliku faili serverprotsessilt.

Blaze viis enda väljatöötatud vahemälustrateegia efektiivsuse hindamiseks läbi hulga simulatsioone, varieerides kliendi vahemälu suurust (seda mõõtis ta failide arvuga, mida vahemälu mahutab) ja parameetrit Δ (valik $\Delta = \infty$ vastab traditsioonilisele klient-server mudeli korral kasutatavale vahemälustrateegiale). Simulatsioonide läbiviimisel selgus, et nimevahemälu suurus ei mõjuta saadud tulemusi märkimisväärselt (pole mingit vahet, kas kasutatakse väikest umbes 100Kb suurust või lõpmatu suurusega nimevahemälu). Simulatsioonide lähteandmed olid kogutud 112 sõlmest koosnevast hajusa failisüsteemi Topaz installatsioonist ja 250 kasutajat[16] teenindanud NFSi installatsioonist. Simulatsioonide läbiviimisel eeldas Blaze, et kliendi

[16] Blaze ei anna oma töös [Blaze93] täpset sõlmede arvu.

kirjutamisstrateegiaks on WBOC-strateegia (s.t. modifitseeritud fail saadetakse serverprotsessile faili sulgemisel) ning valideerimisstrateegiana kasutatakse serveri juhitud valideerimise modifikatsiooni. Kui faili modifitseeritakse ja tehtud muudatused serverprotsessile nähtavaks saavad, leiab serverprotsess nimevahemälust selle faili kohta käiva kirje ning saadab vastava teate kõigile kliendimoodulitele, kelle nimed kuuluvad kirjes sisalduvasse alluvate nimekirja. Teate saanud kliendimoodulid leiavad samal viisil failile vastava alluvate nimekirja ning edastavad saadud teate kõigile alluvatele jne.

Et hinnata, kui efektiivne tema väljatöötatud vahemälustrateegia hajusa failisüsteemi skaleeritavuse suurendamisel on, mõõtis Blaze, mitu korda peab serverprotsess simulatsiooni vältel kliendimoodulitele faile, alluvate nimekirju ning failide kooskõlalisust puudutavaid teateid saatma. Samuti mõõtis Blaze, mitu korda simulatsiooni vältel üldse hajusa failisüsteemi sõlmede vahel faile edastatakse. See annab teatava ettekujutuse, kui koormatud peale serverarvuti ja sellega ühendatud kohtvõrgu on hajusa failisüsteemi ülejäänud sõlmed ning kohtvõrgud.

Parameetri Δ vähendamine vähendab tunduvalt serverprotsessi poolt tehtud failiedastuste arvu. Võttes $\Delta = \infty$ asemel $\Delta = 2$, vähenes see Blaze'i mõõtmistulemuste põhjal (128 või rohkem faili mahutavate kliendi vahemälude puhul) alati vähemalt 2 korda, samal ajal kui kogu hajusas failisüsteemis tehtud failiedastuste arv kasvas kõige rohkem ainult 25% võrra [Blaze93]. Kuigi Δ vähendamisel väheneb ka failide kooskõlalisust puudutavate teadete arv, mida serverprotsess kliendimoodulitele saadab, peab serverprotsess $\Delta = 2$ korral kliendimoodulitele tunduvalt sagedamini alluvate nimekirju saatma kui Δ suuremate väärtuste korral, mistõttu $\Delta = 2$ korral saadab serverprotsess kokkuvõttes enim teateid (vt. Tabel 3.12).

Δ	**2**	**4**	**8**	**16**	**32**	**64**	**∞**
NFSi installatsioonist kogutud lähteandmed	23381	13405	12236	13228	13933	14092	14119
Topazi installatsioonist kogutud lähteandmed	11939	3638	758	410	410	410	410

Tabel 3.12 Serverprotsessi poolt kliendimoodulitele saadetud failide kooskõlalisust puudutavate teadete ja alluvate nimekirju sisaldavate teadete arv.

Samas koormab faili saatmine serverarvutit ja sellega ühendatud kohtvõrku tunduvalt enam kui alluvate nimekirja või faili kooskõlalisust puudutava teate saatmine. Simulatsioonide vältel kliendimoodulitele saadetud failide keskmine suurus oli 11Kb, samal ajal kui näiteks $\Delta = 2$ korral on alluvate nimekirjas kõigest kaks nime. Võib juhtuda, et serveri vahemälus ei ole kõiki vajaliku faili blokke ning puuduvad blokid tuleb serverarvuti kettalt lugeda, mis nõuab täiendavalt serverarvuti protsessoriaega. Blaze'i tehtud mõõtmised näitasid, et 14Kb suuruse faili kliendimoodulile saatmine nõuab üle 10 korra enam serverarvuti protsessoriaega kui alluvate nimekirja või faili kooskõlalisust puudutava teate saatmine [Blaze93]. Seetõttu oletas Blaze, et tänu failiedastuste arvu tunduvale vähenemisele on serverarvuti ja serverarvutiga ühendatud kohtvõrk Δ väikeste väärtuste korral kokkuvõttes vähem koormatud ning hajus failisüsteem on skaleeritavam [Blaze93].

Selle hüpoteesi kinnitamiseks lõi Blaze enda väljatöötatud vahemälustrateegiat kasutava hajusa failisüsteemi prototüübi, mis kannab nime PFS (Parasitic File System), ning testis seda kahe testi abil. Testimiseks loodud PFSi installatsioon koosnes 7 klient- ja 1 serverarvutist. Kumbki test asetab installatsioonile koormuse, mis vastab simulatsioonide läbiviimiseks ühest hajusast failisüsteemist kogutud lähteandmetele. Simulatsiooni lähteandmetes esinevad klientarvutid "jagati" installatsiooni kuuluva 7 klientarvuti vahel; igas klientarvutis töötav testprogramm jäljendas talle "eraldatud" umbes 1/7 klientarvutite tööd. Oma vahemälustrateegia efektiivsuse hindamiseks mõõtis Blaze serverarvuti protsessori koormust ja testi läbimise aega (testi läbimise ajaks luges ta selle testprogrammi täitmise aega, mille täitmine kõige rohkem aega nõuab), võttes algul $\Delta = \infty$ ja siis $\Delta = 2$. Osutub, et võrreldes juhuga $\Delta = \infty$ kulub juhul $\Delta = 2$ mõlema testi puhul serverarvuti protsessoriaega 2 korda vähem ning mõlema testi läbimiseks kulub 2 korda vähem aega [Blaze93]. Kuigi $\Delta = 2$ korral peavad ka kliendimoodulid klientprotsesse teenindama ning klientarvutite koormus seetõttu kasvab, väheneb suuremas hajusas failisüsteemis serverarvuti koormus niipalju, et klientprotsesside töökiirus ei kahane, vaid hoopis kasvab. $\Delta = \infty$ korral on klientarvutid küll vähem koormatud, kuid suuremas hajusas failisüsteemis kulub suur osa klientprotsesside tööajast ülekoormatud serverarvutis töötava serverprotsessi järel ootamisele.

3.6.3 xFSi vahemälustrateegia

1993. aastal alustati Berkeley Ülikoolis hajusa failisüsteemi xFS loomist. Esialgsete kavade kohaselt pidi xFSist saama hajus failisüsteem, mis oleks võimeline koondama endasse tuhandeid sõlmi ning paljude organisatsioonide kohtvõrke [Wang93]. Hiljem xFSi autorite plaanid muutusid ning süsteem loodi keskkonna jaoks, mis koosneb kiirete kohtvõrkude abil ühendatud sõlmedest [And95]. Selles alapunktis vaadeldakse vahemälustrateegiat, mida kavatseti xFSi puhul algselt kasutada.

xFSi vahemälustrateegia kohaselt püütakse vajalikke andmeid kõigepealt kliendi vahemälust hankida. Kui neid seal ei ole, hangitakse andmed mõne teise klientarvuti juures asuvast kliendi vahemälust. Eelmise kahe vahemälustrateegia korral mängis serverprotsess hajusas failisüsteemis küllalt olulist rolli, kuna tema valduses olid alati kõik andmed. Selles alapunktis tutvustatava vahemälustrateegia korral see enam nii ei ole - kõik andmed asuvad klientarvutite juures paiknevates kliendi vahemäludes, serverprotsessi valduses on ainult info, millistes kliendi vahemäludes milliseid andmeid parajasti hoitakse. Peale hajusa failisüsteemi skaleeritavuse suurendamise eesmärgi motiveeris xFSi autoreid taolist vahemälustrateegiat välja töötama ka asjaolu, et seoses kõvaketaste ja mälude hinna langusega on jõutud olukorrani, kus tavalisi tööjaamu varustatakse sellise hulga ketta- ja operatiivmäluga, mis varem oli iseloomulik ainult serverarvutitele [Dahlin94a].

Oma käsitluses eeldasid xFSi autorid, et kliendi vahemälu koosneb nagu AFSi puhulgi klientarvuti kettal asuvast andmevahemälust, mille elementideks on failid, ning klientarvuti operatiivmälus asuvast staatusvahemälust. Staatusvahemälu sisust tehakse perioodiliselt klientarvuti kettale varukoopia. xFS erineb AFSist nelja unikaalse omaduse poolest[17] [Dahlin94a]:

- **Uus vahemälustrateegia** - kui vajalikku faili kliendi vahemälus ei leidu, pöördub kliendimoodul serverprotsessi poole. Serverprotsess edastab nõude ühele sellistest kliendimoodulitest, kes faili parajasti kliendi vahemälus hoiab. Nõude saanud kliendimoodul saadab faili otse nõude esitanud kliendimoodulile. Kui klientprotsess loob või kustutab faili, informeerib kliendimoodul sellest serverprotsessi, et

[17] xFSi autorid esitavad oma töös [Dahlin94a] vaid failide kohta käiva kirjelduse.

serverprotsessi valduses olev info hajusas failisüsteemis eksisteerivate failide kohta oleks täielik.

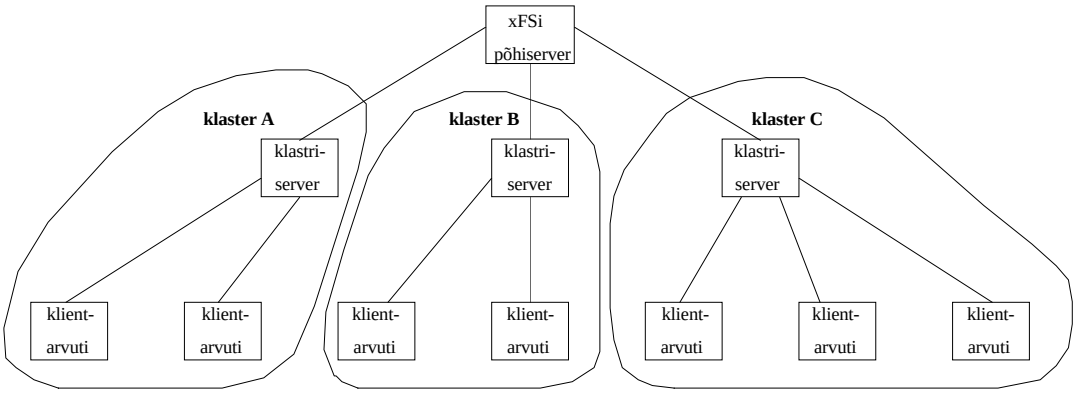
- **Kirjutamisstrateegia puudumine** - kui klientprotsess faili modifitseerib, ei saada kliendimoodul faili tehtud muudatusi serverprotsessile, vaid salvestab need kliendi vahemällu. Seejärel informeerib kliendimoodul serverprotsessi, et faili on modifitseeritud. See xFSi omadus on tingitud xFSi puhul kasutatavast vahemälustrateegiast, mille kohaselt asuvad kõik andmed kliendi vahemäludes ning serverprotsessi valduses on vaid informatsioon andmete asukoha kohta. Sellise lähenemise juures on oluline, et asendamisstrateegia rakendamisel ei jäetaks kliendi vahemälust välja faili viimast kogu hajusas failisüsteemis leiduvat koopiat. Selle vältimiseks märgib kliendimoodul iga kliendi vahemälus paikneva faili, kui klientprotsess faili kirjutab. Kui asendamisstrateegia kohaselt tuleb märgitud fail kliendi vahemälust välja jätta, saadab kliendimoodul selle teisele juhuslikult valitud kliendimoodulile ning informeerib serverprotsessi faili asukoha muutusest. Teine kliendimoodul kohtleb saadud faili, justkui oleks fail talle tema enda nõudmisel saadetud (näiteks LRU-asendamisstrateegia kasutamisel paigutatakse faili nimi LRU-nimekirja algusse).

Need kaks xFSi unikaalset omadust tähendavad, et serverarvuti ja sellega ühendatud kohtvõrgu koormus on palju väiksem kui klient-server mudelil põhinevates hajusates failisüsteemides, sest serverprotsess ei pea enam tegelema andmete kliendimoodulitele saatmise ning neilt modifitseeritud andmete vastuvõtmisega. See tähendab, et xFS on tunduvalt skaleeritavam kui suvaline klient-server mudelil põhinev hajus failisüsteem.

- **Kirjutamisõigused** - xFS kasutab serveri juhitud vahemälu valideerimist. Kui klientprotsess faili kirjutab, siis faili sulgemisel informeerib kliendimoodul sellest serverprotsessi. Serverprotsess edastab vastava teate kõigile ülejäänud kliendimoodulitele, kes seda faili kliendi vahemälus hoiavad. Pärast seda tekib olukord, kus faili ainuke kehtiv koopia asub klientarvuti juures, kus töötab (või töötas) faili modifitseerinud klientprotsess. Serverprotsess eeldab nüüd, et selle klientarvuti juures asuv kliendimoodul omab edaspidi faili suhtes *kirjutamisõigust* - kui klientarvutis töötav klientprotsess faili modifitseerib, ei pea kliendimoodul serverprotsessi enam sellest informeerima (sel pole mõtet, sest üheski teises kliendi vahemälus pole selle faili kehtivat koopiat). Kliendimoodul kaotab kirjutamisõiguse

siis, kui mõnes teises klientarvutis töötav klientprotsess faili avab (kliendimoodul saab selle kindlaks teha, sest faili avamisel pöördutakse faili omandamiseks tema poole), või siis, kui kliendimoodul saab serverprotsessilt teate faili modifitseerimise kohta (võib juhtuda, et teises klientarvutis töötav klientprotsess avab faili kirjutamiseks enne ja sulgeb faili pärast seda, kui kliendimoodul kirjutamisõiguse omandab).

- **Klasterdamine** (*clustering*) - xFSi korral koondatakse klientarvutid *klastriteks* (*clusters*) (vt. Joonis 3.13). Üks klaster moodustatakse nendest klientarvutitest, mis kuuluvad ühele organisatsioonile ning paiknevad võrgutopoloogia mõttes üksteisele lähedal (näiteks on ühendatud ühe kohtvõrgu abil). Igas klastris asub üks serverarvuti ehk *klastriserver* (*cluster server*). Klastriserveri juures töötava serverprotsessi valduses on info, millised andmed milliste antud klastrisse kuuluvate klientarvutite juures asuvad. Kui kliendi vahemälus vajalikku faili pole, pöördub kliendimoodul klastriserveri juures töötava serverprotsessi poole. Kui fail asub mingi antud klastrisse kuuluva klientarvuti juures, edastab serverprotsess nõude selle klientarvuti juures asuvale kliendimoodulile; vastasel korral pöördub ta põhiserveri juures töötava serverprotsessi ehk põhiserverprotsessi poole. Põhiserverprotsessi valduses on info ainult selle kohta, millistest klastrites milliseid andmeid hoitakse. Selle põhjal edastab ta teate vastava klastriserveri juures töötavale serverprotsessile, kes teate omakorda õigele kliendimoodulile edasi saadab.



Joonis 3.13 xFS.

Klasterdamine aitab vähendada info hulka, mida põhiserverprotsess peab haldama. Põhiserverprotsess peab andmete asukohta puudutavat info meeles pidama mitte iga hajusasse failisüsteemi kuuluva klientarvuti kohta, vaid ainult klastrite kohta. Teiseks vähendab klasterdamine teadete arvu, mida põhiserverprotsess peab saatma ja vastu võtma, sest mingis klastris töötav klientprotsess kasutab peaaegu alati samas klastris asuvaid andmeid [Spas93]. Seetõttu suudavad klastriserverid paljusid nõudeid ise rahuldada, edastades nõude mõnele samas klastris asuvale kliendimoodulile. Kliendi vahemälude valideerimisel peab põhiserverprotsess ainult siis sekkuma, kui modifitseeritakse mitmes klastris asuvat faili (kui fail asub ainult ühes klastris, on klastril faili suhtes kirjutamisõigus ning põhiserverprotsessi pole vajadust informeerida). Kolmandaks vähendab klasterdamine teadete ja failide hulka, mida laivõrgu kaudu edastatakse, sest suuremas hajusas failisüsteemis võivad kaks klastrit või klaster ja põhiserver võrgutopoloogia mõttes üksteisest võrdlemisi kaugel paikneda. See on oluline, sest teadete või andmete edastamine laivõrgu kaudu on ajalises ja võrgu koormuse mõttes sageli märksa kallim kui nende edastamine kohtvõrgu kaudu.

xFSi ja AFSi skaleeritavuse võrdlemiseks viisid xFSi autorid läbi mitu simulatsiooni. Nad valisid võrdlemiseks AFSi, sest seda peetakse üheks kõige skaleeritavamaks hajusaks failisüsteemiks, mis on klient-server mudelile üles ehitatud. Simulatsioonide läbiviimiseks vajalikud lähteandmed koguti 237 klient-, 1 serverarvutist ja 4 kohtvõrgust koosnevast NFSi installatsioonist (serverarvuti on ühendatud kõigi kohtvõrkudega). Simulatsioonide läbiviimisel tehti järgmised eeldused:

- klientarvuti kettal paikneva andmevahemälu suurus on 100Mb.
- iga klientarvuti operatiivmälus asub 2048 elementi mahutav staatusvahemälu.
- serveri vahemälu suurus on 128Mb (see eeldus on vajalik ainult AFSi simuleerimisel).
- iga simuleeritud kohtvõrk moodustab ühe klastri; 4 klastriserverit on põhiserveriga ühendatud eraldi kohtvõrgu abil (see eeldus on vajalik ainult xFSi simuleerimisel).
- hajusa failisüsteemi sõlme võimsus vastab DECstation 5000/200 omale.
- kohtvõrkude läbilaskevõime vastab 10Mbit/s Etherneti omale.

xFSi autorid leidsid, et xFSi korral kulub klientprotsesside teenindamisele 6 korda vähem serverarvuti protsessoriaega kui AFSi korral (vt. Tabel 3.13). Kui AFSi

modifitseerida nii, et kasutataks xFSi vahemälustrateegiat ning kirjutamisstrateegia puuduks, kuluks serverarvuti protsessoriaega üle 2 korra vähem (vt. Tabel 3.13).

	Kulunud serverarvuti protsessoriaeg
AFS	1,000
+ uus vahemälustrateegia ja kirjutamisstrateegia puudumine	0,418
+ kirjutamisõigused	0,337
+ klasterdamine (xFS)	0,153

Tabel 3.13 Kulunud serverarvuti protsessoriaeg (näitajad on normaliseeritud AFSi korral kulunud protsessoriaja suhtes).

xFSi autorid tundsid samuti huvi, kui suur on xFSi korral võrgu koormus. Osutub, et xFSi puhul edastatakse võrgu kaudu üle 2 korra vähem infot (andmeid ja teateid) kui AFSi puhul (AFS - 15431Mb, xFS - 7474Mb). xFSi korral on võrgu koormus palju väiksem peamiselt kirjutamisstrateegia puudumise tõttu [Dahlin94a].

	AFS	**xFS**
Põhiserveriga vahetatud info hulk	15431Mb	58Mb
Klastrite vahel edastatud info hulk	-	2902Mb
Klastrite siseselt edastatud info hulk	-	4514Mb

Tabel 3.14 Võrgu koormuse iseloom AFSi ja xFSi korral.

xFSi puhul muutub lisaks veel võrgu koormuse iseloom (vt. Tabel 3.14) - alla 1% infost liigub klastrite ja põhiserveri vahel ning üle 60% infost klastrite siseselt. Kui hajusa failisüsteemi sõlmed on ühendatud laivõrgu abil, on xFS kasutamiseks märksa sobivam kui AFS, sest suur osa infost edastatakse kiireid ja odavaid ühendusteid pidi (kohtvõrgu kaudu).

Lõpuks huvitas xFSi autoreid veel, kui palju klientarvutite koormus xFSi korral tõuseb, kuna kliendimoodulid täidavad lisaülesandeid, mis klient-server mudelil põhinevas hajusas failisüsteemis kuuluksid serverprotsessile. Osutub, et klientarvutite protsessorite koormus on praktiliselt samasugune nagu AFSi puhul [Dahlin94a]. Põhjusena näevad xFSi autorid asjaolu, et kuigi klientarvuti protsessori koormus kasvab teistes klientarvutites töötavate klientprotsesside teenindamise tõttu, väheneb

selle koormus kirjutamisstrateegia puudumise tõttu (kliendimoodul ei saada modifitseeritud faile nende sulgemisel enam serverprotsessile, mis nõuab omajagu klientarvuti protsessoriaega).

3.6.4 Kooperatiivsed vahemälustrateegiad

Kooperatiivsed vahemälustrateegiad töötati välja Berkeley Ülikoolis, ajendatuna võrgutehnoloogia kiirest arengust viimase 2-3 aasta jooksul [Dahlin94b]. Kooperatiivsete vahemälustrateegiate korral püütakse vajalikke andmeid kõigepealt kliendi vahemälust hankida. Kui andmeid seal ei ole, siis otsitakse neid algul serveri vahemälust ning seejärel teise klientarvuti juures asuvast kliendi vahemälust (või vastupidi). Kui ka see ebaõnnestub, pöördutakse lõpuks serverarvuti ketta poole. Kooperatiivsed vahemälustrateegiad on mõeldud hajusate failisüsteemide jaoks, mille sõlmed on ühendatud kiirete võrkude (näiteks ATM- või Myrinet-võrkude) abil. Nende strateegiate korral asuvad kliendi vahemälud klientarvutite operatiivmäludes, sest kiirete võrkude kasutuselevõtmisega on märgatavalt suurenenud vahe andmete kettalt lugemisele kuluva aja ning andmete üle võrgu edastamisele kuluva aja vahel. Seetõttu nõuab kiirete võrkude korral teise klientarvuti kettal asuvate andmete omandamine umbes 10-20 korda rohkem aega kui teise klientarvuti operatiivmälus asuvate andmete omandamine, samal ajal kui aeglasemate võrkude (näiteks 10Mbit/s Ethernet) korral on see vahe vaid 2-3 kordne [Dahlin94b].

Dahlin, Wang, Anderson ja Patterson vaatlevad oma töös [Dahlin94b] lähemalt nelja kooperatiivset vahemälustrateegiat. Nad eeldavad oma käsitluses, et kliendi vahemälu koosneb ainult andmevahemälust ning selle elementideks on failiblokid.

Otsese koopereerumise (*direct client cooperation*) strateegia korral lubatakse *aktiivse* klientarvuti juures asuval kliendimoodulil kasutada *passiivse* klientarvuti operatiivmälu lokaalse kliendi vahemälu laiendusena. Kui aktiivse klientarvuti juures asuvast kliendi vahemälust tuleb asendamisstrateegia kohaselt blokk välja jätta, saadetakse see mõne passiivse klientarvuti juures asuvale kliendimoodulile, kes paigutab saadud bloki vahemällu, justkui oleks see blokk talle tema enda nõudmisel saadetud. Edaspidi on ümberpaigutatud blokk kättesaadav nii aktiivse kui passiivse klientarvuti juures asuvatele kliendimoodulitele. Selle vahemälustrateegia kasutamisel

peavad olema defineeritud kriteeriumid klientarvuti aktiivsuse ja passiivsuse määratlemiseks, samuti peab kliendimooduli jaoks leiduma algoritm, mille abil suvalise klientarvuti aktiivsust või passiivsust kindlaks teha. Otsese koopereerumise strateegia eeliseks on selle lihtsus - seda saab kasutada ka traditsioonilistes klient-server mudelil põhinevates hajusates failisüsteemides ilma serveritarkvara modifitseerimata. Strateegia puuduseks on asjaolu, et kui vajalik blokk asub teise klientarvuti juures asuvas vahemälus, kuid kliendimoodul pole blokki ise sinna paigutanud, pole ta bloki asukohast teadlik ja peab selle omandamiseks ikkagi serverprotsessi poole pöörduma.

Ahne edastamise (*greedy forwarding*) strateegia korral haldab serverprotsess infot kliendi vahemälude sisu kohta. Erinevalt eelmisest vahemälustrateegiast moodustavad kõik kliendi vahemälud *globaalse vahemälu*, mida iga kliendimoodul saab vabalt kasutada. Kui klientprotsessile vajalikku blokki kliendi vahemälus ei ole, pöördub kliendimoodul selle omandamiseks serverprotsessi poole. Nõude saamisel kontrollib serverprotsess, kas nõutud blokk on serveri vahemälus. Kui see nii on, saadetakse blokk nõude esitanud kliendimoodulile. Vastasel korral leiab serverprotsess kliendi vahemälude sisu kajastava info põhjal mõne antud blokki vahemälus hoidva kliendimooduli ja edastab nõude sellele. Nõude saanud kliendimoodul saadab bloki nõude esitanud kliendimoodulile. Kui sellist kliendimoodulit ei leidu, loetakse blokk serverarvuti kettalt serveri vahemällu ja saadetakse nõude esitanud kliendimoodulile. Ahne edastamise strateegia eeliseks on *õiglus* (*fairness*) - iga kliendimoodul võib erinevalt eelmisest strateegiast lokaalset kliendi vahemälu omal äranägemisel (ehk *ahnelt*) hallata, kliendimoodulitel puudub võimalus blokkide paigutamiseks teistesse kliendi vahemäludesse. Strateegia puuduseks on *andmete dubleerimise* oht - ühed ja samad blokid võivad sisalduda mitmes erinevas kliendi vahemälus, mis raiskab asjatult globaalset vahemälu.

Tsentraalse koordineerimise (*centrally coordinated caching*) strateegia erineb ahne edastamise strateegiast selle poolest, et kõik kliendimoodulid loovutavad teatud osa (näiteks 80%) lokaalsest kliendi vahemälust serverprotsessile, kes haldab nii tekkinud globaalset vahemälu nagu serveri vahemälu laiendust. Kui klientprotsessile vajalikku blokki kliendi vahemälus ei ole, pöördub kliendimoodul selle omandamiseks serverprotsessi poole. Nõude saamisel kontrollib serverprotsess, kas nõutud blokk asub serveri vahemälus. Kui see nii on, saadetakse blokk nõude esitanud

kliendimoodulile. Vastasel korral kontrollitakse, kas blokk on paigutatud mõne kliendi vahemälu osasse, mida serverprotsess serveri vahemälu laiendusena haldab. Kui see nii on, edastatakse nõue vastavat kliendi vahemälu haldavale kliendimoodulile, vastasel korral loetakse blokk serverarvuti kettalt ja saadetakse nõudjale. Serverprotsess kasutab globaalse vahemälu puhul sama asendamisstrateegiat nagu serveri vahemälu puhulgi. Kui näiteks LRU-asendamisstrateegia kasutamisel tuleb mingi blokk serveri vahemälust välja jätta, leitakse globaalse vahemälu blokk, millele viimati kõige kauem aega tagasi viidati, ning kliendimoodul, mille juures paiknevas kliendi vahemälus see blokk asub. Serveri vahemälust väljajäetav blokk saadetakse seejärel antud kliendimoodulile, kes asendab globaalse vahemälu vanima bloki saadud blokiga. Serverprotsess paigutab serveri vahemälust väljajäetud bloki nime globaalse vahemälu LRU-nimekirja algusse. Tsentraalse koordineerimise strateegia põhiliseks eeliseks on andmete dubleerimise vältimine, mistõttu iga üksiku kliendimooduli seisukohalt mahub teistesse kliendi vahemäludesse rohkem kiiresti kättesaadavaid andmeid ning tõuseb *globaalne tabamustegur* (globaalse vahemälu tabamustegur). Samas võib *lokaalne tabamustegur* (kliendi vahemälu tabamustegur) langeda, kuna kliendimoodulil puudub võimalus kogu kliendi vahemälu omal äranägemisel hallata.

N-võimalusega edastamise (*N-chance forwarding*) strateegia on teatud mõttes kompromiss ahne edastamise ja tsentraalse koordineerimise strateegiate vahel - kuigi kliendimoodul ei pea kliendi vahemälu teatud osa serverprotsessile loovutama, ei lubata tal kliendi vahemälu siiski täielikult omal äranägemisel hallata. N-võimalusega edastamise ainus erinevus ahnest edastamisest on *üksikutele blokkidele* (*singlets*) osaks saav erikohtlemine. Üksikuks nimetatakse sellist blokki, mis asub vaid ühes kliendi vahemälus. Blokki, mis asub rohkem kui ühes kliendi vahemälus, nimetatakse *dubleerituks* (*duplicated*). N-võimalusega edastamise korral püütakse vältida üksikute blokkide globaalsest vahemälust väljajätmist. Selle strateegia puhul sisaldab iga blokk peale andmete ka loendurit, mis võib omandada täisarvulisi väärtusi lõigust [0, n].

Kui kliendimoodul paigutab bloki kliendi vahemällu, omistab ta bloki loendurile väärtuse null. Kui blokk asendamisstrateegia kohaselt kliendi vahemälust välja jäetakse, pöördub kliendimoodul serverprotsessi poole, et kontrollida, kas tegu on üksiku blokiga (serverprotsess haldab infot kõigi kliendi vahemälude sisu kohta ja suudab kindlaks teha, kas blokk asub veel mõnes kliendi vahemälus või mitte). Kui väljajäetav

blokk on üksik, omistab kliendimoodul selle loendurile väärtuse *n*, saadab bloki juhuslikult valitud kliendimoodulile ning teatab bloki asukoha muutumisest serverprotsessile. Bloki saanud kliendimoodul paigutab selle kliendi vahemällu, justkui oleks see blokk talle tema enda nõudmisel saadetud. Kui blokk tuleb asendamisstrateegia kohaselt ka sellest kliendi vahemälust välja jätta, vähendab kliendimoodul bloki loenduri väärtust ühe võrra ja saadab bloki omakorda uuele kliendimoodulile. Kui loenduri väärtuse vähendamise tagajärjel saab see võrdseks nulliga, siis vahemälust väljajätmisel blokki enam teisele kliendimoodulile edasi ei saadeta.

Kui klientprotsess viitab üksikule blokile, mis asub lokaalses kliendi vahemälus, omistab kliendimoodul bloki loendurile väärtuse null ning jätab bloki vahemällu edasi. Kui teises klientarvutis töötav klientprotsess viitab üksikule blokile, mille loendur ei võrdu nulliga, siis jäetakse see pärast teise arvuti juures asuvale kliendimoodulile saatmist kliendi vahemälust välja. Parameeter N näitab, mitu korda on üksikul blokil lubatud ühest kliendi vahemälust teise liikuda, ilma et talle oleks vahepeal viidatud. Võttes N = 0, osutub N-võimalusega edastamine ahne edastamise strateegiaks.

Sellise vahemälustrateegia rakendamisel võib tekkida olukord, kus vahemälust väljajäetav üksik blokk saadetakse teisele kliendimoodulile, kes sellele blokile ruumi tegemiseks jätab kliendi vahemälust omakorda välja mingi üksiku bloki ja edastab selle kolmandale kliendimoodulile jne. Doominoefekti vältimiseks tuleb kliendi vahemälu asendamisstrateegiat nii modifitseerida, et sellistel juhtudel asendataks üksiku bloki asemel mõni dubleeritud blokk (LRU-asendamisstrateegia kasutamisel näiteks see dubleeritud blokk, millele viimati kõige kauem aega tagasi viidati) [Dahlin94b].

Käsitletud nelja kooperatiivse vahemälustrateegia efektiivsuse hindamiseks viisid nende autorid läbi hulga simulatsioone, võrreldes strateegiaid klient-server mudelil põhinevates hajusates failisüsteemides kasutatava traditsioonilise vahemälustrateegiaga. Simulatsioonide läbiviimiseks vajalikud lähteandmed koguti 42 klient- ja 1 serverarvutist koosnevast hajusa failisüsteemi Sprite installatsioonist. Simulatsioonide läbiviimisel tehti järgmised eeldused:

- kliendi vahemälu on 16Mb suurune ja serveri vahemälu 128Mb suurune.
- kliendi ja serveri vahemälu elementideks on 8Kb suurused blokid.
- kliendi vahemälu kirjutamisstrateegiana kasutatakse läbikirjutamist.

- kliendi vahemälu valideerimisstrateegiana kasutatakse serveri juhitud valideerimist.
- nii kliendi kui serveri vahemälu asendamisstrateegiana kasutatakse LRU-asendamisstrateegiat.
- hajusa failisüsteemi sõlmed on ühendatud võrgu abil, mille läbilaskevõime on 155Mbit/s (tüüpiline ATM-võrkude korral [Coul94]).
- 8Kb suuruse bloki lugemine operatiivmälust nõuab 250 mikrosekundit, selle lugemine kettalt 14800 mikrosekundit.

Otsese koopereerumise strateegia korral eeldati, et passiivsete klientarvutite juures asuvad kliendimoodulid kasutavad neile saadetud blokkide mahutamiseks täiendavat 16Mb suurust vahemälu (tegelikkuses paigutatakse need blokid kliendi vahemällu). Seetõttu on selle strateegia jaoks saadud tulemused paremad kui nad tegelikult peaksid olema. Tsentraalse koordineerimise strateegia korral eeldati, et iga kliendimoodul eraldab serverprotsessile 80% kliendi vahemälust. N-võimalusega edastamise korral eeldati, et N = 2.

Vahemälustrateegiate autorid hindasid kõigepealt seda, kui efektiivsed on nende poolt väljatöötatud strateegiad klientprotsesside töökiiruse suurendamisel. Selleks mõõtsid nad iga strateegia korral kõigepealt keskmist *reaktsiooniaega* (*response time*) - kui kaua klientprotsessil ühe bloki lugemine keskmiselt aega võtab (vt. Tabel 3.15). Samuti mõõtsid nad, kui suur osakaal on vajaliku bloki hankimisel lokaalsel kliendi vahemälul, serveri vahemälul, teise klientarvuti juures asuval vahemälul ja serverarvuti kettal (vt. Tabel 3.16).

Traditsiooniline vahemälustrateegia	**Otsene koopereerumine**	**Ahne edastamine**	**Tsentraalne koordineerimine**	**N-võimalusega edastamine**
2,75 ms	2,62 ms	2,24 ms	1,68 ms	1,59 ms

Tabel 3.15 Bloki lugemisele keskmiselt kulunud aeg erinevate vahemälustrateegiate korral.

	Traditsiooniline vahemälu-strateegia	Otsene koopereerumine	Ahne edastamine	Tsentraalne koordineerimine	N-võimalusega edastamine
Lokaalne kliendi vahemälu	78%	78%	78%	64%	77%
Serveri vahemälu	6,3%	4%	6,5%	16%	6,5%
Teise klientarvuti juures asuv kliendi vahemälu	-	3,5%	3,5%	12,4%	8,8%
Serverarvuti ketas	15,7%	14,5%	12%	7,6%	7,7%

Tabel 3.16 Erinevate vahemälude osakaal andmete hankimisel. Esimene rida näitab, kui suur osa bloki lugemiseks esitatud nõuetest rahuldatakse antud vahemälustrateegia korral kliendi vahemälu põhjal; teine rida näitab, kui suur osa nõuetest rahuldatakse serveri vahemälu põhjal; kolmas rida näitab, kui suur osa nõuetest rahuldatakse teise klientarvuti juures asuva kliendi vahemälu põhjal; neljas rida näitab, kui suur osa nõuetest rahuldatakse serverarvuti ketta põhjal.

Mõõtmistulemused näitavad, et keskmine reaktsiooniaeg on suurem nende vahemälustrateegiate korral, kus serverarvuti kettal on klientprotsessidele vajalike blokkide hankimisel suurem osakaal. See on loomulik tulemus, sest andmete lugemine serverarvuti kettalt nõuab palju rohkem aega kui nende lugemine serverarvuti või teise klientarvuti operatiivmälust. Mõõtmistulemustest nähtub ka, et keskmine reaktsiooniaeg on märgatavalt väiksem nende strateegiate korral, mis üritavad globaalse vahemälu sisu koordineerida, püüdes vältida blokkide dubleerimist või üksikute blokkide vahemälust väljajätmist (tsentraalne koordineerimine ja N-võimalusega edastamine).

Otsese koopereerumise, ahne edastamise ning traditsioonilise vahemälustrateegia korral on kliendi vahemälude keskmine tabamustegur ühesugune (78%), sest kõigi nende strateegiate puhul haldab kliendimoodul lokaalset kliendi vahemälu omal äranägemisel (otsese koopereerumise puhul tehti eeldus, et passiivsete klientarvutite juures asuvad kliendimoodulid kasutavad neile saadetud blokkide mahutamiseks eraldi vahemälu). Tsentraalse koordineerimise strateegia korral on kliendi vahemälu tabamustegur madalam (64%), sest siin on kliendimoodulid sunnitud 80% oma vahemäludest serverprotsessile loovutama. Samas on serveri vahemälu ja globaalse vahemälu tabamustegurid niivõrd kõrged, et ainult 7,6% juhtudest tuleb klientprotsessile vajalik blokk serverarvuti kettalt lugeda. N-võimalusega edastamise strateegia korral on kliendi vahemälude keskmiseks tabamusteguriks 77%, mis näitab,

et üksikute blokkide vahemälus hoidmise kohustus ei mõjuta tabamustegurit kuigi palju.

Kuigi keskmine reaktsiooniaeg on kõigi kooperatiivsete vahemälustrateegiate korral väiksem kui traditsioonilise vahemälustrateegia puhul, tundsid kooperatiivsete vahemälustrateegiate autorid huvi ka selle vastu, kas leidub selliseid klientprotsesse, mille töökiirus nende strateegiate kasutamisel väheneb. Otsese koopereerumise ja ahne edastamise puhul ei vähene ühegi klientprotsessi töökiirus, sest iga kliendimoodul saab kogu oma vahemälu omal äranägemisel hallata (otsese koopereerumise puhul tehti eeldus, et passiivsete klientarvutite juures asuvad kliendimoodulid kasutavad neile saadetud blokkide mahutamiseks eraldi vahemälu). Tsentraalse koordineerimise korral aeglustub ainult ühe klientprotsessi töö 19% võrra, N-võimalusega edastamise korral ei aeglustu ühegi klientprotsessi töö nimetamisväärselt [Dahlin94b]. See näitab, et kuigi tsentraalse koordineerimise ja N-võimalusega edastamise strateegiad vähendavad mingil määral kliendi vahemälu tabamustegurit, kompenseerib selle globaalse vahemälu tabamusteguri kasv.

Lisaks mõõtsid kooperatiivsete vahemälustrateegiate autorid kõigi strateegiate korral serverarvuti koormust, hindamaks, kui efektiivsed on kooperatiivsed vahemälustrateegiad hajusa failisüsteemi skaleeritavuse suurendamisel. Serverarvuti koormuse mõõtmiseks kasutati lihtsustatud mudelit - arvestati ainult teadete saatmisel ja vastuvõtmisel ning blokkide serverarvuti kettalt lugemisel tekkivat koormust, ignoreerides kirjutamisstrateegiast, faili atribuute puudutavatest jms. nõuetest tingitud koormust, kuna see suurendaks serverarvuti koormust iga strateegia korral ühesuguse suuruse võrra. Vahemälustrateegiate autorid eeldasid, et teate vastuvõtmine ja saatmine nõuab 1 koormusühiku, bloki lugemine serverarvuti kettalt 2 ning bloki saatmine 3 koormusühikut.

Tsentraalse koordineerimise strateegia korral serverarvuti koormus suureneb (vt. Tabel 3.17), sest kliendi vahemälude tabamustegurid vähenevad ja serverprotsessile saadetakse rohkem nõudeid kui teiste strateegiate korral. See näitab, et kooperatiivsete vahemälustrateegiate puhul on peale globaalse tabamusteguri tõstmise oluline ka see, et kliendi vahemälude tabamustegurid poleks liiga madalad. Kõik ülejäänud kooperatiivsed vahemälustrateegiad vähendavad serverarvuti koormust ning suurendavad hajusa failisüsteemi skaleeritavust mõõdukalt (vt. Tabel 3.17).

Traditsiooniline vahemälustrateegia	Otsene koopereerumine	Ahne edastamine	Tsentraalne koordineerimine	N-võimalusega edastamine
100%	87%	89%	110%	87%

Tabel 3.17 Serverarvuti koormus kooperatiivsete vahemälustrateegiate kasutamisel (võrreldes juhuga, kus kasutatakse traditsioonilist vahemälustrateegiat).

Otsese koopereerumise strateegiat on küll väga lihtne realiseerida (serveritarkvara ei nõua mingeid muudatusi), kuid selle strateegia puhul ei pruugi aktiivse klientarvuti juures töötav kliendimoodul passiivsetelt klientarvutitelt nii palju operatiivmälu omandada, et keskmine reaktsiooniaeg märgatavalt väheneks ja klientprotsesside töökiirus seetõttu kasvaks. Mõõtmistulemused näitasid, et kui aktiivse klientarvuti juures asuval kliendimoodulil õnnestub lokaalset kliendi vahemälu passiivsete klientarvutite operatiivmälu arvel 25% võrra laiendada, siis väheneb reaktsiooniaeg vaid 1% võrra. Et see väheneks 40%, peab kliendimoodul lokaalset kliendi vahemälu 5 korda laiendama [Dahlin94b]. Kahjuks on uurimused näidanud, et klientarvutid on passiivsed tüüpiliselt vaid lühema aja vältel. See tähendab, et enne, kui kliendimoodul jõuab passiivselt klientarvutilt omandatud operatiivmällu suuremal arvul blokke paigutada, muutub passiivne klientarvuti aktiivseks ja tema juures asuv kliendimoodul täidab kogu sealse kliendi vahemälu selles klientarvutis töötavatele klientprotsessidele vajalike blokkidega.

Ahne edastamise strateegiat tuleb pidada väga heaks vahemälustrateegiaks, sest tal pole otseseid puudusi ja samas on ta võrdlemisi efektiivne. Ühest küljest on teda võrdlemisi lihtne realiseerida, teisest küljest on ta õiglane (võimaldab kliendimoodulitel kliendi vahemälusid omal äranägemisel hallata) ning suurendab klientprotsesside töökiirust (keskmine reaktsiooniaeg on umbes 20% võrra väiksem kui traditsioonilise vahemälustrateegia puhul), suurendamata seejuures serverarvuti koormust. Seda vahemälustrateegiat kasutab hajusa failisüsteemi xFS 1995. aastal valminud prototüüp [And95].

Tsentraalse koordineerimise strateegia efektiivsuse määrab see, kui suur osa igast kliendi vahemälust globaalse vahemälu moodustamiseks eraldatakse. Serverprotsessile eraldatud osa suurendamisel kasvab globaalne tabamustegur ja vähenevad kliendi vahemälu tabamustegurid ning serverprotsessile eraldatud osa

vähendamisel on vastupidi. Mõõtmistulemused näitavad, et keskmine reaktsiooniaeg on vähim siis, kui igast kliendi vahemälust loovutatakse serverprotsessile 40-90%. Vahemälustrateegiate autorid pidasid sobivaimaks 80% vahemälu eraldamist, sest sellise valiku puhul saavutatakse parimad tulemused ka muude kliendi vahemälu suuruste ja töökoormuste korral kui autorid simuleerisid [Dahlin94b].

N-võimalusega edastamise strateegia kasutamisel on keskmine reaktsiooniaeg vähim. Ka serverarvuti on selle strateegia puhul vähe koormatud, sest erinevalt tsentraalse koordineerimise strateegiast suurendab N-võimalusega edastamine globaalset tabamustegurit nii, et kliendi vahemälude tabamustegurid märkimisväärselt ei vähene. Strateegia autorid tegid kindlaks, et keskmine reaktsiooniaeg väheneb kõige rohkem, kui N = 0 asemel võtta N = 1. Kui N = 1 asemel võtta N = 2, saavutatakse veel mõningane keskmise reaktsiooniaja vähenemine, kuid parameetri N suuremad väärtused enam erilist täiendavat efekti ei anna [Dahlin94b].

Summary

This thesis is about distributed file systems. The theory of distributed file systems is complex and contains many subfields. Therefore this thesis focuses only on caching in a distributed file system while other issues (replication, security etc.) are considered briefly.

This thesis consists of three chapters. Chapter 1 introduces some basic definitions, requirements that a modern distributed file system should satisfy and a model of a file service. Chapter 2 describes two world's most popular distributed file systems. NFS (originally designed by Sun Microsystems) has now become an industrial standard supported by most operating system vendors. AFS is known as one of the most scalable distributed file systems in widespread use today. Chapter 3 is the main part of this thesis and focuses on caching issues. Nowadays most researchers agree that caching is mandatory part of a distributed file system design. There is no known distributed file system today which does not use this techinque to improve system's performance and scalability. Because of its importance I decided to choose caching as the main thesis topic.

This thesis is based on recent Ph.D. theses, books, articles and technical reports. Since there were more than 30 different sources involved, it was not easy to compose an in-depth study which would reflect all different views and opinions. Although I probably didn't achieve this goal in perfect, I hope that reading this thesis helps its reader to understand more about distributed file systems.

Kasutatud kirjandus

[And95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang, “Serverless Network File Systems”, *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995

[Baker91] Mary G. Baker, John K. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, “Measurements of a Distributed File System”, *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, lk. 198-212, 1991

[Blaze93] Matthew Addison Blaze, “Caching in Large-Scale Distributed File Systems”, *Ph.D. Thesis (Technical report TR-93-397)*, Princeton University, 1993

[Call95] B. Callaghan, B. Pawlowski, P. Staubach, “NFS Version 3 Protocol Specification”, *Technical report RFC-1813*, Network Working Group, 1995

[Cao94a] Pei Cao, Edward W. Felten, and Kai Li, “Application-Controlled File Caching Policies”, *Technical report TR-94-445*, Princeton University, 1994

[Cao94b] Pei Cao, Edward W. Felten, and Kai Li, “Implementation and Performance of Application-Controlled File Caching”, *Technical report TR-94-462*, Princeton University, 1994

[Cao96] Pei Cao, “Application-Controlled File Caching and Prefetching”, *Ph.D Thesis (Technical report TR-96-522)*, Princeton University, 1996

[Coul94] George Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems: Concepts and Design*, Addison-Wesley, 1994

[Dahlin94a] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. “A Quantitative Analysis of Cache

Policies for Scalable Network File Systems", *Technical report CSD-94-798*, University of California at Berkeley, 1994

[Dahlin94b] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, David A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance", *Technical report CSD-94-844*, University of California at Berkeley, 1994

[Froese95] Kevin W. Froese, Richard B. Bunt, "The Effect of Client Caching on File Server Workloads", *Technical report*, University of Saskatchewan, 1995

[Gray90] Cary G. Gray and David R. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency", *Technical report CS-90-1298*, Stanford University, 1990

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems, Vol.6, No.1*, lk. 51-81, 1988

[Kleim86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Technical Report*, Sun Microsystems, 1986

[Kumar94] Puneet Kumar, "Mitigating the Effects of Optimistic Replication in a Distributed File System", *Ph.D. Thesis (Technical report CS-94-215)*, Carnegie Mellon University, 1994

[Muntz92] D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash", *Proceedings of the Winter USENIX Conference*, lk. 305-313, 1992

[Nelson87] Michael Nelson, Brent Welch, John Ousterhout, "Caching in the Sprite Network File System", *Technical report CSD-87-345*, University of California, Berkeley, 1987

[Nelson88] Michael Newell Nelson, "Physical Memory Management in a Network Operating System", *Ph.D. Thesis (Technical report CSD-88-471)*, University of California at Berkeley, 1988

[Oust85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, “A Trace-Driven Analysis of the UNIX 4.2 BSD File System”, *Technical report CSD-85-230*, University of California, Berkeley, 1985

[Sand85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, Bob Lyon, “Design and Implementation of the Sun Network Filesystem”, *Proceedings of the Summer USENIX Conference*, lk. 119-130, 1985

[Sat89] M. Satyanarayanan, “Integrating Security in a Large Distributed System”, *ACM Transactions on Computer Systems, Vol.7, No.3*, lk. 247-280, 1989

[Sat93] M. Satyanarayanan, “Distributed File Systems”, *Distributed Systems* (ed. Sape Mullender), Addison-Wesley, 1993

[Shir92] Ken W. Shirriff, John K. Ousterhout, “A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System”, *Proceedings of the Winter USENIX Conference*, lk. 315-331, 1992

[Siegel92] Alexander Siegel, “Performance in Flexible Distributed File Systems”, *Ph.D. Thesis (Technical report TR92-1266)*, Cornell University, 1992

[Silb94] Abraham Silberschatz, Peter B. Galvin, *Operating System Concepts*, Addison-Wesley, 1994

[Spas93] Mirjana Spasojevic, M. Satyanarayanan, “A Usage Profile of a Wide-Area Distributed File System”, *Technical report CS-93-207*, Carnegie Mellon University, 1993

[Sun88] Sun Microsystems, Inc., “RPC: Remote Procedure Call Protocol Specification”, *Technical report RFC-1057*, Network Working Group, 1988

[Sun89] Sun Microsystems, Inc., “NFS: Network File System Protocol Specification”, *Technical report RFC-1094*, Network Working Group, 1989

[Taylor86] Bradley Taylor, David Goldberg, “Secure Networking in the Sun Environment”, *Technical report*, Sun Microsystems, Inc., 1986

[Thom87] James Gordon Thompson, “Efficient Analysis of Caching Systems”, *Ph.D. Thesis (Technical report CSD-87-374)*, University of California, Berkeley, 1987

[Wang93] Randolph Y. Wang and Thomas E. Anderson, “xFS: A Wide Area Mass Storage File System”, *Technical report CSD-93-783*, University of California, Berkeley, 1993

[Will92] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt, “Disk Cache Replacement Policies for Network Fileservers”, *Technical report*, University of Saskatchewan, 1992